Solving Practical Problems in Datacenter Networks

by

Xin Wu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Bruce Maggs

_____
Jeffrey Chase

_____
Romit Roy Choudhury

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2013

ABSTRACT

Solving Practical Problems in Datacenter Networks

by

Xin Wu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Bruce Maggs

_____
Jeffrey Chase

_____
Romit Roy Choudhury

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2013

Abstract

The soaring demands for always-on and fast-response online services have driven modern datacenter networks to undergo tremendous growth. These networks often rely on scale-out designs with large numbers of commodity switches to reach immense capacity while keeping capital expenses under check. Today, datacenter network operators spend tremendous time and effort on two key challenges: 1) how to efficiently utilize the bandwidth connecting each source and destination pair and 2) how to promptly handle network failures with minimal disruptions to the hosted services.

To resolve the first challenge, we propose solutions in both network layer and transport layer. In the network layer solution, we propose DARD, a *D*istributed *A*daptive *R*outing architecture for *D*atacenter networks [Wu and Yang (2012)]. DARD allows each end host to reallocate traffic from overloaded paths to underloaded paths without central coordination. We use congestion game theory to show that DARD converges to a Nash equilibrium in finite steps and its gap to the optimal flow allocation is bounded in the order of $\frac{1}{logL}$ with L being the number of links. We use a testbed implementation and simulations to show that DARD can achieve a close-to-optimal flow allocation with small control overhead in practice.

In the transport layer solution, We propose *Explicit Multipath Congestion Control Protocol* (MPXCP), which achieves four desirable properties: fast convergence, efficiency, being fair to flows with different round trip times and negligible queue size. Intensive *ns-2* simulation shows that MPXCP can quickly converge to efficiency and fairness without

building up queues regardless of different delay-bandwidth products.

To resolve the second challenge, recent research efforts have focused on automatic failure localization. Yet, resolving failures still requires significant human interventions, resulting in prolonged failure recovery time. Unlike previous work, we propose NetPilot [Wu et al. (2012)], a system which aims to quickly *mitigate* rather than resolve failures. NetPilot mitigates failures in much the same way operators do – by deactivating or restarting suspected offending components. NetPilot circumvents the need for knowing the exact root cause of a failure by taking an intelligent trial-and-error approach. The core of NetPilot is comprised of an Impact Estimator that helps guard against overly disruptive mitigation actions and a failure-specific mitigation planner that minimizes the number of trials. We demonstrate that NetPilot can effectively mitigate several types of critical failures commonly encountered in production datacenter networks.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Symbols

l_i	a link in a network with index i.
B_i	l_i's capacity.
N_i	number of flows via l_i.
S_i	a link's fair share, B_i/N_i.
P_k	set of paths for the k^{th} flow.
s	strategy, $[p_1, p_2, \ldots, p_n]$, a collection of paths, where $p_k \in P_k$.
$C_p(s)$	path congestion, the smallest link fair share along path p.
$C(s)$	the smallest link fair share in network.

Abbreviations

DCN_p	a production datacenter network
DCN	datacenter network
LAG	link aggregation group
nhops	next hops
ToR	top of rack switch
CORE	core switch
AGG	aggregation switch

Acknowledgements

I would like to thank my advisor Xiaowei Yang for guiding me to explore many exciting aspects of networking. In the first half of my PhD study, she taught me how to find research problems, how to identify the fundamental trade-offs of different design choices and how to systematically reason every aspect of a complex system. In the second part of my PhD study, she gave me as much freedom as possible. Thanks to her trust and generosity, I was able to enrich my industrial experience and dig into practical problems. Every time when I was lost in research or at a decision point, she is always there to help me out.

It was my great honor to work with Ming Zhang from Microsoft Research. He taught me how to identify real problems in production, how to systematically find the right abstraction for those problems and how to transform a theoretical model to a practical solution.

Lihua Yuan and Dave Maltz have been another two great mentors. Their rich network management experience, creative ideas and great mentoring skill have shaped my taste of networking systems. They also helped me with all possible resources to conduct my crazy testbed evaluation. Without Lihua and Dave's coaching and help, I could never have reached this far.

I benefited a lot from collaborations with many professors and colleagues that led to this dissertation. I especially thank Daniel Turner and Chao-Chih Chen for our in-depth collaboration on the system design and prototyping in Chapter 4. I also would like to thank Bruce Maggs, Jeff Chase and Romit Choudhury for being part of my Research Ini-

tiate Project, Preliminary Exam and PhD Dissertation committees and giving constructive feedback.

I would also like to thank many other collaborators on various projects, including Hongqiang Liu, Roger Wattenhofer, Yu Chen, Dong Xiang, Zao Liu, Kyle Farlow, Daniel Constantine, Thanos Vasilakos. It has been an exciting experience to work with you.

Thanks very much to Marilyn Butler for her selfless and extremely patient help with my numerous VISA and curriculum planning problems. I also thank all my lab mates, including Michael Sirivianos, Xin Liu, Ang Li, Qiang Cao, Yang Chen, Jun Hu, Bingyang Liu, Jannie Tan, Chi-Yao Hong, Chunyi Peng and Hongyi Zeng, for their thoughtful comments on my research.

I benefited a lot from my Microsoft internships. Many thanks to Randy Kern, Varugis Kurien, Chuanxiong Guo, Yongqiang Xiong, Srikanth Kandula, Ratul Mahajan, Chao Zhang, Long Zhang, Dongluo Chen and Huoping Chen at Microsoft for giving me hands-on industrial internship experiences.

I would like to thank my wife Wei Sun for sharing her life with me and I would like to thank my parents Jinjun and Lina for their support. I would also like to thank all my friends for bringing happiness to my life.

This dissertation work was supported by NSF awards CNS-0845858 and CNS-1040043.

1

Introduction

According to an authoritative Internet trend report published in 2013 [Meeker and Wu (2013)], the cloud computing market will continue to grow nearly 50% in the next year, which then becomes a $25 billion market. Driven by this explosive increase in demand for cloud applications, modern datacenters can easily have tens to hundreds of thousands of servers. For example, Google has at least 16 significant datacenters all around the world. Its total server number in 2011 is around 9 million [Hoelzle and Barroso (2009)]. To interconnect the large number of servers and also to facilitate both inter and intra datacenter traffic, a large number of switches, routers and other devices form a complex datacenter network (DCN). How to design a scalable, manageable and economical datacenter network is a challenging problem. More specifically, 1) how to efficiently utilize the bandwidth connecting each source and destination pair and 2) how to promptly handle network failures with minimal disruptions to the hosted services are on top of the challenge list. This thesis is dedicated to address these two challenges.

1.1 From Scale-up to Scale-out Datacenter Network Architecture

Over the past decade, datacenter network design went, roughly, through two stages: scale-up architecture and scale-out architecture. The overall trend is: people rely more and more on commodity hardware and complex software to manage datacenter networks.

1.1.1 Traditional Scale-up Datacenter Network Architecture

In Ethernet, hosts are allocated with MAC addresses and the switches forward Ethernet frames based on their destination MAC addresses. To forward traffic, Ethernet switches essentially have two functions. First, they cooperate to construct a spanning tree to provide loop-free paths. Second, if a switch does not have a forwarding entry for a frame's destination MAC address, it floods that frame over the entire spanning tree. A switch also learns how to reach a MAC by remembering the source MAC's incoming interface.

However, the above Ethernet protocols do not scale to large networks: MAC addresses are flat and thus cannot be aggregated. Also, the spanning tree protocol wastes a large amount of bandwidth by blocking switch ports. To overcome these two limitations, traditional datacenter networks are designed as follows: a large number of small Ethernet islands are interconnected by routers. Each Ethernet island is assigned with one or more IP subnets. Packets among Ethernet islands are forwarded by layer-3 protocols.

The best design practice for traditional datacenter architecture is described in the Cisco Datacenter Infrastructure Design Guide [Cisco Systems, Inc. (2011)]. The signature of this design is that by moving up to a higher hierarchy, the switches become higher speed and denser in number of ports. The advantage of this design is obvious: all devices and protocols are off-the-shelf and reliable. However, the highest density switches at the top of the hierarchy are expensive and limit the network size. As far as we know, the state of the art nonblocking 10Gbps switch has only 128 ports, on which it is almost impossible to support large datacenters as announced by Google and Microsoft.

1.1.2 Existing Scale-out Datacenter Network Architectures

One practical solution to resolve the above scalability limitation is to expand the topology into a multi-rooted tree to provide multiple equal-cost paths between any host pair [Al-Fares et al. (2008); Greenberg et al. (2009)]. Figure 2.1 shows a 3-stage multi-rooted tree topology. The topology has three vertical layers: *Top-of-Rack* (*ToR*), *aggregation* (*agg*), and *core*. A *pod* is a replicable management unit inter-connected by the cores. However, this scale-out topology itself does not solve the scalability problem because existing routing and transport protocols have limited support for multiple paths at the scale of a datacenter. As a result, both the industry and academia have been actively looking for solutions to efficiently and fairly utilize the network capacity. At a high-level view, there are two categories of solutions: distributed flow-to-path assignment and centralized flow-to-path assignment.

Equal Cost Multipath (ECMP) is a practical solution of random flow-to-path assignment. ECMP-enabled switches hash flows based on flow identifiers to multiple equal-cost next hops. This design paradigms improve the bisection bandwidth, but can still cause permanent hash collisions on congested links [Al-Fares et al. (2010)].

VL2 [Greenberg et al. (2009)] is another solution running in production. It uses ToR switches to forward flows to randomly selected cores to approximate packet-level valiant load balancing (VLB) [Lakshman et al. (2004)]. Operators prefer flow-level VLB over packet-level VLB to prevent potential packet reordering. It achieves similar bisection bandwidth as ECMP [Al-Fares et al. (2010); Wu and Yang (2012)].

Multipath TCP [Wischik et al. (2011)] views the datacenter capacity as a resource pool and tries to efficiently and fairly utilize this resource pool by extending TCP's AIMD concept into multipath scenarios. This approach can potentially penetrate the existing industry practice of datacenter architecture design. In the network layer, routing's goal is no longer to compute the least costly path but to enumerate all the paths connecting any

host pair. In the transport layer, a source and destination pair can simply split one flow along all the paths using MPTCP.

Hedera [Al-Fares et al. (2010)] introduces a centralized controller to approximate an optimal flow-to-path assignment given dynamic traffic load. The software-defined-network community has provided increasingly reliable infrastructures for the centralized controlling [Openflow (2008)]. Hedera achieves close-to-optimal flow allocation

1.1.3 Proposed Practical Datacenter Architectures

In this thesis, we propose two solutions in both network layer and transport layer to efficiently utilize the bandwidth connecting each source and destination pair in the scale-out topology.

In the network layer solution, we advocate conservatively making incremental improvement to datacenter networks based on existing reliable routing technologies. We propose *DARD*, a *D*istributed *A*daptive *R*outing architecture for *D*atacenter networks [Wu and Yang (2012)]. DARD allows each end host to reallocate traffic from overloaded paths to underloaded paths without central coordination.

In the transport layer solution, we take a clean slate approach and propose *MPXCP*, Explicit Multipath Congestion Control Protocol, which leverages both multipath TCP and explicit congestion control. MPXCP can quickly converge to efficiency and fairness without building up queues despite different delay-bandwidth products.

1.2 From Failure Diagnosis to Failure Mitigation in Datacenter Networks

Given the large number of network devices in datacenters, failures become the norm rather than the exception. In the second half of this thesis, we will describe both our measurement and system work about datacenter network failure recovery. A well-accepted definition of a network failure is defined by pre-existing measurement frameworks. There are two types of such frameworks: passive measurement and active measurement. A passive measure-

ment framework logs network events such as link down or switch reboot. The most common practice is syslog [Lonvick (2001)]. An active measurement framework proactively measures certain indexes that can reflect the running state of the network. For example, if a keep-alive message between two servers fails, either the servers or the network connecting those servers are in an abnormal state.

1.2.1 Traditional Three-step Failure Recovery Procedure

Traditionally, network operators follow a three-step procedure to react to network failures: 1) detection; 2) diagnosis; and 3) repair. Diagnosis and repair are often time-consuming, because the sources of failures vary widely, from faulty hardware components to software bugs to configuration errors. Operators must sift through many possibilities just to narrow down potential root causes. These diagnosis and repair sometimes require third-party device vendors' assistance, further lengthening the failure recovery time. Because of the above challenges, it can take a long time to recover from disruptive failures.

1.2.2 Proposed Four-step Failure Recovery Procedure

Realizing the problem above, we take a fundamentally different approach to tackle the failure recovery problem in large-scale DCNs. We advocate a four-step process to react to failures: 1) detection; 2) *mitigation*; 3) diagnosis; and 4) repair. We argue that it is more important to mitigate failures than to fix them in real-time. Here "mitigate" means taking actions that alleviate the symptoms of a failure, possibly at the cost of temporarily reducing spare bandwidth or redundancy. Timely and effective failure mitigation enables a DCN to operate continuously even in the presence of failures, and allows operators to dive directly into failure diagnosis and repair.

We present **NetPilot**, an *automated system* that adopts our four-step process to *quickly* mitigate failures in a large-scale DCN. We identify and address two key technical challenges. First, automatic failure mitigation carries substantial risk because a mitigation

action may have a severe consequence and compromise the health of the entire network. NetPilot prevents this problem by accurately predicting the impact of mitigation actions and executing them within a pre-defined safety guardrail. Second, an excessive number of trials will unnecessarily lengthen failure disruption periods. NetPilot avoids this problem by ordering the possible mitigation actions based on their likelihood of success and potential impact.

The remainder of this thesis is organized as follows: Chapter 2 presents DARD, the practical adaptive routing architecture. Chapter 3 describes MPXCP, the explicit multipath transport protocol. Chapter 4 shows NetPilot, the automated network failure mitigation system. Chapter 5 summarizes this thesis.

2

DARD: A Practical Distributed Adaptive Routing Architecture for Datacenter Networks

2.1 Introduction

Deploying applications in datacenters has become a well-accepted trend because it is economical and efficient. Modern applications usually require intensive inter-server communication, *e.g.*, MapReduce and indexing. Driven by these applications, the bisection bandwidth demand is undergoing an explosive growth. To achieve this goal under budget, today's datacenter network often leverages commodity switches to form a multi-rooted tree to provide multiple equal-cost paths between any host pair [Al-Fares et al. (2008); Greenberg et al. (2009)]. Figure 2.1 shows a 3-stage multi-rooted tree topology. The topology has three vertical layers: *Top-of-Rack* (*ToR*), *aggregation* (*agg*), and *core*. A *pod* is a replicable management unit inter-connected with the cores.

However, existing routing and transport protocols have limited support for multipath at the scale of a datacenter. As a result, both the industry and academia have been actively looking for solutions to efficiently and fairly utilize the network capacity. At a high-level view, there are two types of mechanisms to explore the path diversities in dat-

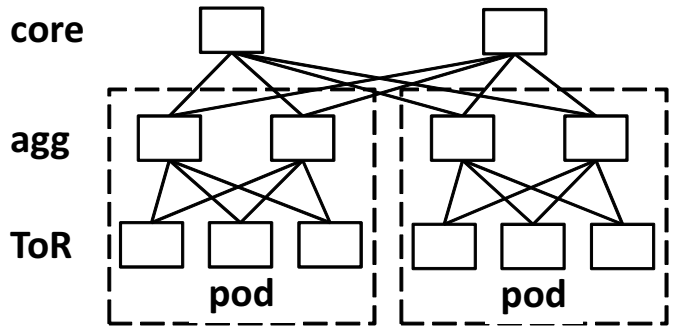


FIGURE 2.1: A multi-rooted tree topology for a datacenter network.

acenters: centralized flow-to-path assignment, which leverages a centralized controller to approximate the optimal flow allocation, and random flow-to-path assignment, in which switches hash flows based on flow identifiers to multiple equal-cost paths. Both of these design paradigms improve the bisection bandwidth. However, each has its limitations. The centralized approach introduces a potential scaling bottleneck [Curtis et al. (2011)]. The random flow allocation causes permanent hash collisions on congested links [Al-Fares et al. (2010)].

Despite the disadvantages of the above solutions, their key concepts have been deployed or are about to be deployed in production datacenters. We believe that the fundamental reason that makes these solutions practical is because all of them are conservatively making incremental improvement based on existing reliable technologies. In addition, We abstract four design guidelines for practical datacenter architectures, including *reliability*, *capability of improving bisection bandwidth*, *scalability* and *debugging-friendly*. We will introduce these four guidelines and explain the pros and cons of existing datacenter architectures in detail in Chapter 2.2.

This chapter presents DARD, a readily deployable routing architecture that enables end hosts to independently allocate dynamic flows. We aim to make DARD practical by strictly following the above four design guidelines. The key design challenge is how to distributively achieve close-to-optimal load balancing with low overhead. To address

this challenge, each end host runs a selfish flow allocation algorithm that demonstrably converges to a Nash equilibrium with bounded gap to the optimal in finite steps. We implement a DARD prototype on testbed and a DARD simulator on *ns-2*. Further, We use static traffic patterns to show that DARD can quickly converge to a stable state. We also use dynamic traffic patterns to show that the bisection bandwidth gap between DARD's flow allocation and the optimal flow allocation is small.

To the best of our knowledge, DARD is the first distributed adaptive routing architecture for datacenters. Our contributions are:

1. We justify four design guidelines for practical datacenter architectures and explain the pros and cons of existing solutions.

2. We design DARD, which is proven to converge to a Nash equilibrium in finite steps with bounded gap to the optimal flow allocation.

3. We conduct intensive evaluation in both testbed and simulation to show DARD's quick convergence, high bisection bandwidth, and small control overhead.

The rest of this chapter is organized as follows. We explain four design guidelines for practical datacenter architectures in Chapter 2.2. Chapter 2.3 describes DARD's design in detail. In Chapter 2.4, we prove that DARD converges to a Nash equilibrium in finite steps and its gap to the optimal flow allocation is bounded. In Chapter 2.5, we introduce how we implement the system in both testbed and simulator. We evaluate DARD in Chapter 2.6. Chapter 2.7 concludes our work.

2.2 Guidelines for Practical Datacenter Architectures

From the operational perspective, a **practical** datacenter architecture solution should take at least four aspects into consideration.

First of all, the solution should be **reliable**. Being reliable is always the number one requirement of datacenter operation. As a result, operators have a very strong incentive to rely on well-proven technologies. They prefer to conservatively and incrementally improve off-the-shelf solutions rather than to aggressively deploy any advanced technology with potentially buggy code.

On top of being reliable, as long as the solution can fulfill the bandwidth requirement from the applications, it is a good candidate. The more **capable of improving bisection bandwidth**, the more likely the solution will be deployed.

Third, the solution should be **scalable**, *i.e.*, it can be easily replicated to an upgraded datacenter despite the datacenter's scale. Otherwise, every datacenter upgrading may cause a fundamental change to the already-proven architecture, which is an unacceptable operation overhead.

The last aspect, which has been significantly negligent, is **debugging-friendly**. Network debugging itself is challenging enough [Handigol et al. (2012)]. More devices in datacenters imply more failures and bugs, which has become a significant operational overhead [Wu et al. (2012)]. Counter-intuitively, the datacenter's highly symmetric architecture does not simplify the network debugging. The key reason is: a datacenter network provides too many possibilities for a packet to traverse. Without the capability to trace the exact path a packet takes, it is very difficult to localize a problem.

Given all the above four guidelines for practical datacenter network designs, it becomes understandable why operators prefer some architectures over others. In the rest of this section, we will review the state of the art datacenter network architectures to understand their fundamental advantages and limitations.

Equal-Cost-Multi-Path forwarding (ECMP) [Hopps (2000)] is one known solution running inside production datacenters [Cisco Systems, Inc. (2011)]. ECMP-enabled switches hash flows based on flow identifiers to multiple equal-cost next hops. It is reliable because all its building blocks have been proven in production: TCP, intra and inter domain routing

protocols and consistent hashing. Even though it causes permanent collision and degrades the bisection bandwidth [Wu and Yang (2012)], it has become the default solution when advanced architectures fail [Al-Fares et al. (2010)]. ECMP scales reasonably well. To the best of our knowledge, switches supporting 64-way ECMP has been off-the-shelf. However, ECMP is not debugging-friendly. Without knowing the exact hushing function on each switch and without the capability of logging the packet headers in real time, it is very difficult to trace a packet. Hence, end-to-end ping and traceroute become less useful.

VL2 [Greenberg et al. (2009)] is another solution running in productions. It uses ToR switches to forward flows to randomly selected cores to approximate packet-level valiant load balancing (VLB) [Lakshman et al. (2004)]. Again, operators prefers flow-level VLB over packet-level VLB to prevent potential packet reordering. It achieves similar bisection bandwidth as ECMP [Al-Fares et al. (2010); Wu and Yang (2012)]. However, it significantly improves the scalability compared with ECMP, because tunneling packets to randomly selected cores only depends on the widely available encapsulation/decapsulation feature. Operators can easily expand a datacenter in the horizontal direction without worrying about the limited ECMP capability. VL2 does not facilitate network debugging either, because tracking the path taken by a packet requires logging the randomly selected core in real time, which is not trivial in practice.

Hedera [Al-Fares et al. (2010)] introduces a centralized controller to approximate an optimal flow-to-path assignment given dynamic traffic load. The software-defined-network community has provided increasingly reliable infrastructures for the centralized controlling [Openflow (2008)]. Hedera achieves close-to-optimal flow allocation and is also debugging friendly because the controller knows every detail about the flow allocation. However, Hedera's major concern is its scalability. [Curtis et al. (2011)] and [Benson et al. (2010)] systematically analyze the overhead of centralized flow allocation and implies that Hedera needs parallelism route computation to support realistic datacenter traffic patterns. Besides, we will show in our evaluation that Hedera's control overhead is pro-

portional to the number of flows, which is not bounded. With the expanding of the size of a datacenter and the explosive increase of traffic demand, how to scale Hedera is a challenging problem.

Multipath TCP [Wischik et al. (2011)] views the datacenter capacity as a resource pool and tries to efficiently and fairly utilize this resource pool by extending TCP's AIMD concept into multipath scenarios. This approach can potentially penetrate the existing industry practice of datacenter architecture design. In the network layer, routing's goal is no longer to compute the least costly path but to enumerate all the paths connecting any host pair. In the transport layer, a source and destination pair can simply split one flow along all the paths using MPTCP. To the best of our knowledge, MPTCP is still not completely standardized in IETF and experimental code has just been released. Considering the legacy TCP has experienced numerous of major improvement even after it was standardized, the conservative industry may not deploy MPTCP in the near future. Given the current status of MPTCP, we can hardly conduct a fair analysis about its scalability and whether it is debugging-friendly.

2.3 DARD Design

As we have explained in Chapter 2.2, none of existing practical datacenter architectures can fulfill the four design guidelines. In this section, we first explain how we make our design choices to ensure that DARD can fulfill all the guidelines. We then describe the DARD design in detail.

2.3.1 Design Choices

We choose the scheduling granularity to be a TCP flow for reliability reasons. The legacy TCP has been the dominant transport protocol in datacenters. Besides, its implementation has been intensively tested in production. As a result, datacenter operators are very conservatively improving TCP [Alizadeh et al. (2010)]. Since packet reordering can significantly

degrade TCP's performance, we choose to allocate one flow to one path to prevent any systematic risk of packet reordering.

We use the term *elephant flow* or *elephant* to refer to a continuous TCP connection transferring bytes more than a certain threshold. We focus on allocating elephant flows to improve bisection bandwidth for two reasons. First, existing measurement shows that 1% of the datacenter flows take more than 90% of all the bytes [Greenberg et al. (2009)]. Consequently, reallocating elephants can significantly improve the aggregated flow rate by preventing permanent collisions [Al-Fares et al. (2010)]. Second, random flow allocation already performs reasonably well for small, short RPC-like flows when their quantity is significantly larger than the number of paths, in which case the hash collisions are averaged over the bottlenecks [Al-Fares et al. (2010); Wu et al. (2012)].

To make DARD scalable to large topologies, we choose the distributed tunneling approach as VL2 such that path diversities are encoded by encapsulation but is not limited by the number of equal cost paths supported by commodity switches. Besides, without centralized coordination, DARD is not possibly to be bottlenecked by the centralized controller.

The key step to design a debugging-friendly datacenter architecture is to make packets traceable, *i.e.*, to trace the exact path a packet takes with little overhead. DARD achieves this goal by encoding the path in an encapsulated IP header. Different from VL2 which encapsulates packets at ToRs, DARD encapsulates packets at hosts. As a result, we can leverage the rich functions provided by end hosts to trace packets. For example, TCPdump can be easily deployed in end hosts but not in commodity switches. Besides, pushing IP routing all the way to the end hosts facilitates network layer debugging tools,*e.g.*, ping and traceroute, to trace every single hop along a path.

2.3.2 Overview

DARD has a conceptual control loop of three steps. First, it detects elephant flows at end hosts. Second, each end host queries the switches for their workload. Third, each end host reallocates flows to relieve congested paths. The purpose of the first two steps is to provide timely information for the third step. We loose this conceptual control loop by making each of the three steps an independent process. Each of the three processes has its own control loop. Since the overhead of elephant detection is small, an end host constantly monitors all the outgoing flows. We use the term *query interval* to refer to the time interval between two consecutive set of queries initiated from the same end host. We use the term *control interval* to refer to the time interval between two consecutive flow reallocations conducted by the same host pair.

For the purpose of easy expression, we introduce the definitions that will be frequently referred to in the rest of this chapter. We use B_l to note link l's capacity. N_l denotes the number of elephant flows on l. We define l's *fair share* $S_l = B_l/N_l$, which is the bandwidth each elephant should be allocated if they fairly share that link ($S_l = 0$, if $N_l = 0$). Link l's *link state* (LS_l) is defined as a triple [B_l, N_l, S_l]. Switch r's *switch state* (SS_r) is defined as $\{LS_l \mid l \in r$'s outgoing links$\}$. A path p refers to a set of links connecting a source and destination ToRs. The first and last hop links are not included in a path because we only consider the situation in which flows are bottlenecked by the network but not the NIC capacity. If link l has the smallest S_l among all the links of path p, we use LS_l to represent p's *path state* (PS_p). This smallest S_l is also referred as path p's *path congestion*.

Figure 2.2 shows DARD's components. A switch in DARD has only two functions: 1) It forwards packets to the next hop according to a pre-configured forwarding table; 2) It tracks its running state and replies to host queries.

An end system has three components. The *elephant detector* constantly monitors all the outgoing flows and considers a flow as elephant once its size grows beyond a threshold.

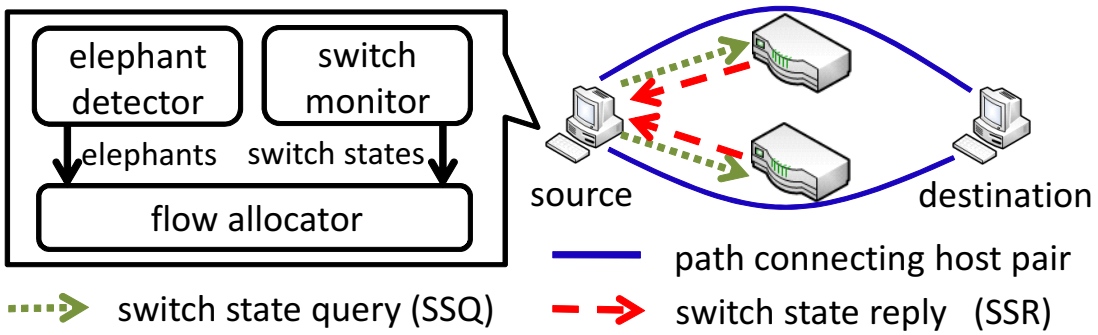


FIGURE 2.2: This figure shows DARD's three components on a host. The *elephant detector* constantly detects elephant flows. The *switch monitor* periodically queries switches for their states. The *flow allocator* periodically reallocates flows from overloaded paths to underloaded paths.

The *switch monitor* periodically sends switch state queries (*SSQ*) to switches and processes the corresponding switch state replies (*SSR*). The *flow allocator* periodically reallocates flows from overloaded paths to underloaded paths.

Since the two switch functions and the elephant detector at an end host are simple, we focus the rest of this chapter on the other three building blocks: DARD's addressing and routing, switch monitor and flow allocator.

2.3.3 Addressing and Routing

In this section, we will introduce how DARD leverages datacenter's hierarchical topology to encode path information into IP address and how DARD achieves the goal of being scalable and debugging-friendly by tunneling.

Figure 2.3 uses a fattree topology to illustrate DARD's addressing and routing. There are four trees in this topology, each of which is rooted from one of the cores, *e.g.*, all the switches and hosts highlighted by the dotted boxes form a tree with its root $core_1$. This strictly hierarchical structure facilitates multipath routing via customized addressing [Al-Fares et al. (2008)]. We borrow the idea from NIRA [Yang et al. (2007)] to split an end-to-end path into *uphill* and *downhill* segments and to encode a path in the source

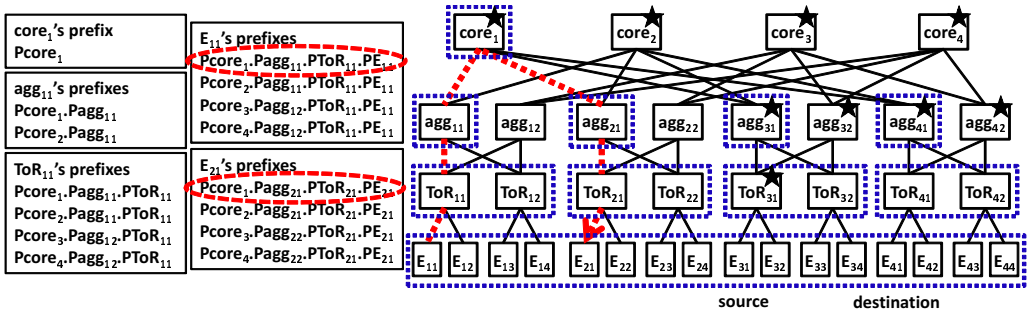


FIGURE 2.3: This figure illustrates DARD's addressing and routing on a fattree topology. The dotted boxes highlight the tree rooted from $core_1$. E_{11} and E_{21}'s addresses highlighted by dotted circles uniquely encode the uphill path $ToR_{11} \rightarrow agg_{11} \rightarrow core_1$ and the downhill path $core_1 \rightarrow agg_{21} \rightarrow ToR_{21}$. The switches highlighted with starts are the switches that E_{31} sends SSQs to when elephant flows are from E_{31} to E_{41}.

and destination addresses. In DARD, each of the core switches obtains a unique prefix and allocates non-overlapping subdivisions of the prefix to its sub-trees. A sub-tree recursively allocates non-overlapping subdivisions of its prefix to lower hierarchies. By this hierarchical address allocation, every device receives multiple IP addresses, each of which represents the device's position in one of the trees.

For example, in Figure 2.3 we use $core_i$ to represent the i^{th} core, agg_{ij} to represent the j^{th} aggregation switch in the i^{th} pod. We follow the same rule to interpret ToR_{ij} for the top of rack switches and E_{ij} for the end hosts. We use the device identifiers prefixed with letter P and delimited by periods to illustrate how prefixes are allocated along the hierarchies. The first core is allocated with prefix $Pcore_1$. It then allocates non-overlapping prefixes $Pcore_1.Pagg_{11}$, $Pcore_1.Pagg_{21}$ $Pcore_1.Pagg_{31}$, $Pcore_1.Pagg_{41}$ to its four sub-trees. The sub-tree rooted from agg_{11} further allocates four prefixes to lower hierarchies.

One advantage of this hierarchical addressing is that one address uniquely encodes the sequence of upper-level switches that allocate the address, *e.g.*, in Figure 2.3, E_{11}'s address $Pcore_1.Pagg_{11}.PToR_{11}.PE_{11}$ uniquely encodes the sequence of address allocation: $core_1 \rightarrow agg_{11} \rightarrow ToR_{11}$. A host address pair can further uniquely identify a path, *e.g.*, in Figure 2.3, we can use the host address pair highlighted by dotted circles to uniquely

Table 2.1: agg_{11}'s forwarding tables.

Downhill Table	
Prefixes	next-hop
$Pcore_1.Pagg_{11}.PToR_{11}$	ToR_{11}
$Pcore_1.Pagg_{11}.PToR_{12}$	ToR_{12}
$Pcore_2.Pagg_{11}.PToR_{11}$	ToR_{11}
$Pcore_2.Pagg_{11}.PToR_{12}$	ToR_{12}
Uphill Table	
Prefixes	next-hop
$Pcore_1$	$core_1$
$Pcore_2$	$core_2$

encode the dotted path from E_{11} to E_{21} via $core_1$. We call the partial path encoded by the source address the *uphill path* and the partial path encoded by the destination address the *downhill path*. To reallocate a flow to a different path, we can simply use different address combinations without reconfiguring any forwarding table.

Each switch maintains a *uphill table* and an *downhill table*. The uphill table keeps the prefix entries for the upstream switches and the downhill table keeps the prefix entries for the downstream switches. Table 2.1 shows agg_{11}'s two tables. To forward a packet, a switch first looks up the destination in the downhill table using the longest prefix matching. If a match is found, the packet will be forwarded to the corresponding downstream switch. Otherwise, the switch will look up the source address in the uphill table for the next-hop upstream switch. A core switch maintains only the downhill table.

Each host in DARD is also assigned a location independent IP address (*ID*), which uniquely identifies the host and is used for making TCP connections. The mapping from IDs to underlying IP addresses is maintained by a DNS system and cached locally. To deliver a packet, the source encapsulates the packet with a source and destination address pair. Switches forward the packet according to the encapsulating header. The destination decapsulates the packet on its arrival.

In fact, the above twice-looking-up forwarding is not necessary for a fattree, because

a core switch in fattree uniquely determines one path for a host pair. However, not all the multi-rooted trees share the same property. For a general datacenter topology, operators can follow a similar addressing schema to allocate prefixes along the hierarchies. In case more IP addresses are assigned to one network interface, we propose to use IP alias to configure multiple IP addresses to one network interface. Modern operating systems support a large number of IP alias to associate with single network interface, *e.g.*, Linux kernel 2.6 sets the limit to be $256K$ IP alias per interface. Windows NT 4.0 has no limitation on this number.

DARD's addressing and routing design is scalable because it achieves multipath routing by encapsulating path information into packet headers at end hosts. This solution is not affected by the topology scale. Encapsulating path information at end host is also debugging-friendly because operators can easily trace the exact path each packet takes.

2.3.4 Switch Monitor

To enable end hosts to adaptively allocate flows, DARD requires a timely channel to inform every host of the network's running state. Compared with letting switches periodically push their running states to hosts, letting hosts actively query switches can potentially decrease the control traffic by on-demand polling. As a result, we design the component of *switch monitor* on every host to actively query switches for their states. This section first describes a straw man design of the switch monitor. Then we decrease the straw man's control traffic by on-demand polling.

The straw man design is straightforward: Each switch tracks its state locally. A host's switch monitor periodically sends switch state queries (*SSQ*s) to every switch and assembles the corresponding switch state replies (*SSR*s) to compute every path's congestion. If every host sends one query to every switch and receives the corresponding replies in one query interval, the straw man's control traffic for that interval can be estimated in formula (2.1), where pkt_size is the packet size of one SSQ plus its corresponding SSR. The

implication is that the straw man's control traffic is bounded by the topology size.

$$num\_of\_hosts \times num\_of\_switches \times pkt\_size \tag{2.1}$$

We observe three optimization opportunities from the above straw man design: 1) If a host is not transferring elephants, it is not necessary to monitor any switch. 2) If a host is transferring elephants, it does not have to query every switch in the topology. As the example shown in Figure 2.3, E_{31} is sending elephants to E_{41}. E_{31} has to monitor only the switches marked by the starts because the rest of the switches are not on any path connecting the source and destination. We do not mark ToR_{41} because its outgoing link to the destination is directly connected to the host NIC. 3) Suppose a switch sw is on the paths from source src to destination dst_1 and is also on the paths from src to destination dst_2, Then one SSR from sw to src can be used for computing the path congestions for both (src, dst_1) and (src, dst_2), two source and destination pairs.

Based on the above three observations, we optimize the straw man by reducing the number of SSQs: 1) a host's switch monitor keeps idle until the host starts to transfer elephants. 2) A source monitors only the switches along the paths to the destination. The switch whose outgoing link is the cut of every path to the destination or is directly connected to the host NIC is not included. 3) The number of SSQs a source sends to a switch in one query interval is at most one.

Our switch monitor's design requires every switch to count flow numbers and to interact with hosts. These two functions are supported by OpenFlow switches.

2.3.5 Flow Allocator

This section describes DARD's last building block, the *flow allocator* running on each end host. It takes the detected elephants and switch states as the input and reallocates flows to relieve overloaded paths. The important observation that motivates our design is: given an elephant's elastic demand and small latencies in datacenters, elephants tend

Algorithm 1: Selfish Flow Allocation

1: S_{max} = maximum S_l in $\{pv[i].S_l\}$;
2: i_{max} = S_{max}'s index in pv;
3: S_{min} = minimum S_l in $\{pv[i].S_l \mid fv[i] > 0\}$;
4: i_{min} = S_{min}'s index in pv;
5: if ($i_{max} == i_{min}$) return;
6: $estimation = \frac{pv[i_{max}].B_l}{pv[i_{max}].N_l+1}$
7: if ($estimation - S_{min} > \delta$)
8: reallocate one elephant from path i_{min} to path i_{max}.

to fully and fairly utilize their bottlenecks. As a result, moving flows from a path with small fair share to a path with large fair share can improve the fairness and increase the minimum flow rate. Chapter 2.4.1 justifies this intuition. Based on the above intuition, DARD's high-level approach is to enable every host to selfishly increase the minimum fair share it observes. Algorithm 1 illustrates a host pair's flow allocation process in one control interval.

In DARD, every source and destination pair maintains two vectors at the source: the *path state vector* (pv), whose i^{th} item is the path state of the corresponding path connecting the source and destination, and the *flow vector* (fv), whose i^{th} item is the number of elephants sent by the source along the corresponding path. Line 6 estimates the fair share of the i_{max}th path if another elephant flow is added to it. The δ in line 7 is a positive threshold to decide whether to reallocate a flow. If we set δ to 0, line 7 ensures that the algorithm increases the minimum fair share observed by the source. If we set δ to be larger than 0, the algorithm stops reallocating flows once the estimation becomes close enough to the observed minimum fair share. We will describe how to choose δ in Chapter 2.6.1 and Chapter 2.4.3. Ideally, If every end host in the datacenter periodically takes this procedure, we expect that given a traffic pattern, every end host can eventually stop reallocating flows, *i.e.*, DARD converges. We also expect that a small δ achieves larger fair share when DARD converges, and a large δ accelerates the convergence process.

Another design aspect to consider is that distributed load-sensitive routing can lead

to permanent path oscillation [Khanna and Zinky (1989)], which can potentially result in degraded bisection bandwidth because TCP may suffer from reordering or even packet loss every time the path changes. Figure 2.4 shows an example. There are three source and destination pairs: (E_1, E_2), (E_3, E_4) and (E_5, E_6). Each of the pairs has two paths and is transferring two elephants. The source in each pair runs Algorithm 1 independently without knowing the others' reactions. In the beginning, the shared path, link $switch_1$-$switch_2$, has no elephant. As a result, each of the three sources reallocates one flow to it and thus, the number of elephants on the shared path increases to three. This triggers the three sources to reallocate flows back to their original paths. This process repeats and causes permanent oscillation and bandwidth under-utilization.

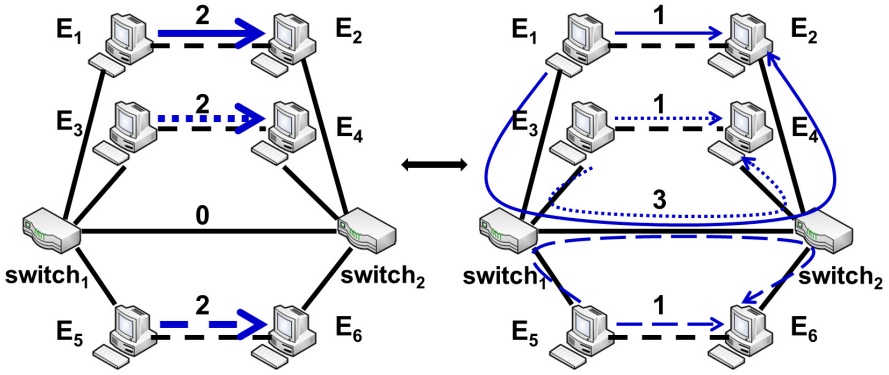


FIGURE 2.4: This figure shows a path oscillation example. Each of the three host pairs has two paths, one of which is shared with the other two pairs. Each pair transfers two elephant flows. If all of the three pairs constantly move flows to a less congested path in synchronization, DARD causes a permanent path oscillation.

The fundamental reason for path oscillation is that different sources independently and thus, inevitably reallocate flows off from the same overloaded path in synchronization. To understand how serious this synchronization problem is, we present a simple analytical model. We would like to estimate how many synchronized source and destination ToR pairs are likely to use the same link from a core to an aggregation switch (this inter-pod link is usually the bottleneck).

In a n-pod fattree, we use X to note the $\frac{n^2}{4}$ cores. Given a link l from a core to

an aggregation switch, there are $\frac{n(n-1)}{2}$ (noted as Y) ToRs outside of the pod that link l connects to. And there are $\frac{n}{2}$ (noted as Z) ToRs inside of the pod that link l connects to. Suppose every ToR randomly chooses one core as the intermediate hop to a ToR in a different pod, the expected number of source and destination ToR pairs using link l is $\frac{YZ}{X} = n - 1$, which is bounded by the fattree size.

Because DARD's distributed design choice makes it difficult to coordinate all the hosts to completely prevent synchronized flow reallocation, we choose to introduce certain randomness to eliminate synchronization. Given the above limited number of synchronized source and destination ToR pairs on a bottleneck link, we believe simply adding randomness to each host's control interval can work fairly well. Our evaluation in Chapter 2.6.3 also shows that randomizing each end host's control interval successfully prevents path oscillation.

Despite DARD's simple design, it is effective. We prove in Chapter 2.4.2 that DARD converges to a Nash equilibrium regardless of different values of δ. We also prove in Chapter 2.4.3 that if we carefully choose δ, DARD's gap to the optimal flow allocation is bounded. We further discover in our evaluation that DARD's performance is not sensitive to the δ value.

2.4 DARD Modeling

In this section, we prove DARD can converge to a Nash equilibrium in finite steps and its gap to the optimal flow allocation is bounded.

2.4.1 Explanation of the objective

We assume TCP is the dominant transport protocol in datacenter, which tries to achieve max-min fairness if combined with fair queuing. Each end host moves flows from overloaded paths to underloaded paths to increase its observed minimum fair share. This section explains that given a static traffic demand and max-min fair bandwidth allocation,

the global minimum fair share is the lower bound of the global minimum flow rate, thus increasing the minimum fair share actually increases the global minimum flow rate.

Theorem 1: Given a static traffic demand and max-min fair bandwidth allocation, the global minimum fair share is the lower bound of global minimum flow rate.

Proof: First we define a bottleneck link according to [Boudec (2000)]. A link l is a bottleneck for a flow f if and only if 1) l is fully utilized, and 2) flow f has the maximum rate among all the flows using l.

A link l_i's fair share S_i is defined as B_i/N_i, where B_i is the link capacity and N_i is the number of flows via l_i. Without loss of generality, we assume link l_0 has the minimum fair share S_0. Flow f has the minimum rate, min_rate. Link l_f is flow f's bottleneck. Theorem 1 claims $min\_rate \geqslant S_0$. We prove it by contradiction.

According to the bottleneck definition, min_rate is the maximum flow rate on link l_f, and thus $B_f/N_f \leqslant min\_rate$. Suppose $min\_rate < S_0$, we get

$$B_f/N_f < S_0 \tag{2.2}$$

which is contradictory with S_0 being the minimum fair share. As a result, $min\_rate \geqslant S_0$, *i.e.*, the minimum fair share is the lower bound of the global minimum flow rate.

The implication of above theorem is that by increasing each source and destimation pair's observed minimum fair share, both the global minimum fair share and the global minimum flow rate increase as well.

2.4.2 Convergence proof

We now formalize DARD's selfish flow allocation algorithm as a *congestion game* [Busch and Magdon-Ismail (2009)] and prove it converges to a Nash equilibrium in finite steps.

We assign an index to each flow of the static traffic demand and use P_k to represent the set of paths for the k^{th} flow. A *Strategy* $s = [p_1, p_2, \ldots, p_n]$ is a collection of paths,

where $p_k \in P_k$ and is delivering the k^{th} flow. Given a strategy s, we use $S_l(s)$ to note link l's fair share. A path p's *path congestion* $C_p(s)$ is represented by the smallest link fair share along that path. The *system congestion* $C(s)$ is the smallest link fair share in the network. Notation s_{-k} refers to the strategy s without p_k, *i.e.*, $[p_1, \ldots, p_{k-1}, p_{k+1}, \ldots, p_n]$. $(s_{-k}, p_{k'})$ refers to the strategy $[p_1, \ldots, p_{k-1}, p_{k'}, p_{k+1}, \ldots, p_n]$. The k^{th} flow is *locally optimal* in strategy s if for any other possible path $p_{k'} \in P_k$.

$$C_{p_k}(s) \geqslant C_{p_{k'}}(s_{-k}, p_{k'}) \tag{2.3}$$

A *Nash equilibrium* is a state where all flows are locally optimal. A strategy $\widetilde{s}$ is *global optimal* if for any strategy s, $C(\widetilde{s}) \geqslant C(s)$. We further define the *price of anarchy* (*PoA*) as follow.

$$PoA = \text{minimum}_{s \in S} \frac{C(s)}{C(\widetilde{s})} \tag{2.4}$$

In other words, PoA quantifies the worst system congestion with DARD's selfish behavior.

Theorem 2: If every source and destination pair follows Algorithm 1 and at most one flow is relocated at any given step, then the global minimum fair share does not decrease in every step and all the source and destination pairs converge to a Nash equilibrium in finite steps.

Proof: A strategy s's *strategy vector* $sv(s)$ is in the form of [v_0,v_1,v_2,...], where v_k is the number of links with fair share between $k\delta$ and $(k+1)\delta$. These links are referred as a *fair share equivalence class*. This δ is the positive parameter in the selfish flow allocation algorithm. As a result, this vector cluster the links into multiple equivalence classes according to their fair share values. $\sum_k v_k$ is the total number of links in the network.

Suppose s and s' are two strategies, $sv(s) = [v_0,v_1,v_2,\ldots]$ and $sv(s') = [v'_0,v'_1,v'_2,\ldots]$. We define $s = s'$ if $v_k = v'_k$ for all $k \geqslant 0$. $s' < s$ if there exists some j such that $v'_j < v_j$

and $\forall k < j, v'_k \leqslant v_k$. It is easy to show that given three strategies s, s' and s'', if $s'' \leqslant s'$ and $s' \leqslant s$, then $s'' \leqslant s$.

Given a specific network, traffic pattern and δ, there are only finite numbers of strategy vectors. According to the definition of " $=$ " and " $<$ ", we can find at least one strategy $\widetilde{s}$ that is the *smallest*, *i.e.*, for any strategy s, $\widetilde{s} \leqslant s$. This $\widetilde{s}$ has the largest minimum fair share or has the least number of links with the minimum fair share, and thus is the global optimal.

If one flow selfishly changes its path and makes the strategy move from s to s', this action decreases the number of links with a particular fair share and increases the number of links with a relatively larger fair share. In other words, $s' < s$. This indicates asynchronous and selfish flow allocation does not decrease global minimum fair share in every step until all flows reach local optimum. Since the number of strategy vectors is finite, the steps to converge to a Nash equilibrium is finite.

Lemma 1: The global optimal strategy $\widetilde{s}$ is also a Nash equilibrium strategy.

Proof: Because $\widetilde{s}$ is the smallest strategy, no flow can have a further movement to decrease $\widetilde{s}$, *i.e.*, every flow reaches local optimum. As a result, this global optimal strategy is also a Nash equilibrium strategy.

In the proof of Theorem 1, one key parameter is δ, the width of a fair share equivalence class. It makes the continuous model discrete, *i.e.*, we treat two fair shares as the same value if they locate in the same equivalence class. The δ value is a trade-off between a larger global minimum fair share when the system converges and a smaller number of convergence steps. A small δ forces a source and destination pair to explore a better flow allocation as long as its observed most congested and least congested links are located in different equivalence classes. On the other hand, a large δ decreases the convergence steps by enlarging each equivalence class. One thing to note is **the system is proven to**

converge to a Nash equilibrium despite the δ value.

2.4.3 Bound the price of anarchy

So far, we have proven if every source and destination pair follows DARD's selfish strategy, the system converges to a Nash equilibrium. We also proved that the optimal flow allocation, which maximizes the global minimum fair share, is also a Nash equilibrium. However, these two equilibriums are not necessarily the same. Next we are going to prove that for some carefully assigned δ, we can bound the gap between DARD's and the optimal equilibriums.

We first assign δ according to (2.5). $C(s)$ represents the system congestion after DARD converges. B_l represents link l's capacity when l is the bottleneck for certain flows. N_l is the number of flows on the bottleneck l. As a result, the left side of (2.5) means **the width of an equivalence class is no larger than the minimum fair share after convergence**, otherwise the system congestion remains in the same equivalence class before and after DARD's scheduling. The right side of (2.5) makes sure the width of an equivalence class is larger than the maximum change of all the bottleneck links' fair shares **in the entire convergence process**, and consequently **every flow reallocation causes bottleneck links to move to no further than the adjacent equivalence classes**.

$$C(s) \geqslant \delta \geqslant \text{maximum } (\frac{B_l}{N_l - 1} - \frac{B_l}{N_l}) \tag{2.5}$$

Suppose the k^{th} flow is locally optimal with its path congestion C_{p_k}, then according to the definition of local optimum, every alternative path for that flow must have congestion at most $C_{p_k} + \delta$, *i.e.*, the bottlenecks of all the alternative paths are either in the same or the adjacent equivalence class, otherwise reallocating this flow to an alternative path will improve the flow's path congestion.

We define a *path-cut* of a source and destination pair to be a set of links such that every path connecting the pair must use at least one link from this set. A straightforward

observation is shown in Lemma 2. We are now ready to use this lemma to bound the price of anarchy in Theorem 3.

Lemma 2: Given a strategy $s = [p_1, p_2, \ldots, p_n]$ where the k^{th} flow is locally optimal, then there is a path-cut for that source and destination pair in which every link's congestion is at most $C_{p_k} + \delta$.

Theorem 3: Given a network topology with P being the longest path length, M being the maximum number of paths between any source and destination, L being the number of links and B_{min} and B_{max} being the minimum and maximum link capacities in the topology, then the price of anarchy defined in (2.4) is bounded by $\frac{B_{min}}{(\frac{logL}{log(M-1)}+1)(M-1)B_{max}P}$.

Proof: As shown in Theorem 2's proof, for any non-Nash equilibrium strategy s', there always exists at least one Nash equilibrium strategy s, such that $s < s'$, *i.e.*, we can treat a Nash equilibrium strategy as a local optimal strategy. As a result, we can further restrict the s in (2.4) to be any Nash equilibrium strategy.

We use $C(s)$ to note the system congestion of a Nash equilibrium strategy s. Let J_0 be the set of links whose link congestion is $C(s)$ ($|J_0|$ is at least 1). Let F_0 be the set of flows that use at least one link in J_0. By Lemma 2, every source and destination pair of the flows in F_0 has a path-cut in which every link's congestion is at most $C(s) + \delta$. Let J_1 be the union of J_0 and all the above path-cuts of every flow in F_0. Then we have $J_0 \in J_1$ and every link in J_1 has the congestion at most $C(s) + \delta$.

We repeat the above process as follows. J_i is the union of J_{i-1} and the path-cuts for the source and destination pairs of the flows in F_{i-1}. By Lemma 2, every link in J_i has the congestion at most $C(s) + i\delta$. Then we construct flow set F_i, each of which uses at least one link in J_i. By the above construction, we obtain a sequence of J_i and F_i, where $J_0 \subseteq J_1 \ldots \subseteq J_i$ and $F_0 \subseteq F_1 \ldots \subseteq F_i$. We stop this process until we find the first set J_k

such that

$$|J_k| \leqslant (M-1)|J_{k-1}| \tag{2.6}$$

where M is the maximum number of paths between any source and destination in the topology. In the next paragraph we explain why this k must exist.

In the above process, if at the m^{th} step we find $|J_m| \leqslant (M-1)|J_{m-1}|$, then $k = m$ and we stop the construction. Otherwise, noticing the congestion upper bound for all the link sets is at most the minimum link capacity B_{min}, we continue the above construction until at the n^{th} step this congestion upper bound reaches B_{min}. Then the difference between J_n and J_{n-1}, noted as D, is the set of links with the congestion between $B_{min} - \delta$ and B_{min}, *i.e.*, **there is only one flow via each link of** D. We use F_D to note the flows via D and thus have $|F_D| \leqslant |D|$. Let M be the maximum number of paths between any source and destination, then in the $(n+1)^{th}$ step, each flow in F_D can contribute at most $(M-1)$ links to construct J_{n+1}, *i.e.*, $|J_n|$ **is expanded less than** $(M-1)$ **times**. As a result, if $k = (n+1)$, then $|J_k| \leqslant (M-1)|J_{k-1}|$.

So far we have shown the k satisfying (2.6) exists. Each link in J_{k-1} has the fair share at most $(C(s) + (k-1)\delta)$. Let T be the times that links in J_{k-1} are used by the flows in F_{k-1} and B_{min} be the minimum link capacity in the topology, then $T \geqslant \frac{B_{min}}{C(s)+(k-1)\delta}|J_{k-1}|$. On the other hand, the links in J_{k-1} are only used by the flows in F_{k-1}, and suppose that the maximum path length is P, then $T \leqslant P|F_{k-1}|$. As a result,

$$kC(s) \geqslant C(s) + (k-1)\delta \geqslant \frac{|J_{k-1}|B_{min}}{|F_{k-1}|P} \tag{2.7}$$

The first "$\geqslant$" is because of the left side of (2.5). Next we are going to bound the right side of (2.7). Since J_k has a path-cut for every flow in F_{k-1} and thus every flow in F_{k-1} must use at least one link in J_k under all the possible strategies, including the optimal $\widetilde{s}$, *i.e.*, links in J_k are used at least $|F_{k-1}|$ times. According to pigeonhole principle, despite

how flows in F_{k-1} are being allocated, at least one link in J_k is used at least $|F_{k-1}|/|J_k|$ times. Let B_{max} be the maximum link capacity in the topology, we have

$$\frac{B_{max}}{C(\widetilde{s})} \geqslant \frac{|F_{k-1}|}{|J_k|} \tag{2.8}$$

By combining (2.8), (2.7) and (2.6) we get

$$\begin{aligned} \frac{C(s)}{C(\widetilde{s})} &\geqslant \frac{|J_{k-1}|B_{min}}{k|F_{k-1}|PC(\widetilde{s})} \\ &\geqslant \frac{|J_{k-1}|B_{min}}{k|J_k|B_{max}P} \\ &\geqslant \frac{B_{min}}{k(M-1)B_{max}P} \end{aligned} \tag{2.9}$$

We are then going to bound k in (2.9). Let L be the number of links in the topology. Because k is the first number that satisfies (2.6), thus

$$L \geqslant |J_{k-1}| \geqslant (M-1)|J_{k-2}| \ldots \geqslant (M-1)^{k-1}|J_0|$$

Since $|J_0| \geqslant 1$, we have

$$\frac{logL}{log(M-1)} + 1 \quad \geqslant \quad k \tag{2.10}$$

By combining (2.9) and (2.10) we bound the price of anarchy in (2.11) which is only related to the topology but not the traffic demand. **It is in the order of** $\frac{1}{logL}$.

$$\begin{aligned} PoA &= \text{minimum}_{s \in S} \frac{C(s)}{C(\widetilde{s})} \\ &\geqslant \frac{B_{min}}{(\frac{logL}{log(M-1)} + 1)(M-1)B_{max}P} \end{aligned} \tag{2.11}$$

2.5 Implementation

To evaluate DARD's performance and to show DARD is practically deployable, we implemented a prototype and deployed it on a 4-pod fattree topology in DeterLab [Lab (2003)]. We also implemented a simulator in *ns-2* to evaluate DARD's performance in large scale.

2.5.1 Testbed

We set up a 4-pod fattree topology using 4-NIC PCs acting as the switches. Their IP addresses are configured according to Section 2.3.3. We enable IP alias [Sironi (2001)] on each end host to support multiple IP addresses on one NIC. All switches run OpenFlow V1.0 on top of the standard Ubuntu 10.04 LTS image. An OpenFlow switch maintains flow and port level statistics and allows us to customize its forwarding tables.

We implement a NOX [Gude et al. (2008)] component to configure all the flow tables during switches initialization. This component allocates the downhill flow table to $table_0$ and the uphill flow table to $table_1$ to enforce a higher priority for the downhill table. All entries are configured to be permanent. NOX is often used as a centralized controller for OpenFlow enabled networks. However, DARD leverages NOX only once to initialize the static flow tables. The same purpose can be achieved by other techniques.

A daemon program runs on every end host. It has the three components shown in Figure 2.2. The elephant detector leverages the TCPTrack [Steve (2005)] at each end host to report an elephant flow if the bytes of a TCP connection grow beyond certain threshold. The switch monitor tracks the fair share of all the equal-cost paths connecting the source and destination ToR switches by querying switches for their states using the interfaces of *aggregated flow statistics* provided by OpenFlow [Openflow (2008)]. The flow allocator moves elephant flows from overloaded paths to underloaded paths according to the selfish flow allocation algorithm. We use the Linux IP-in-IP tunneling as the encapsulation and decapsulation module. We assign every end host with an IP address from a different prefix

as its ID. The mappings from IDs to the corresponding NIC IP addresses are kept in a file at all end hosts.

We also implement the ECMP algorithm on every switch. It decides the next hop by hashing the source and destination NIC IP addresses and TCP ports using CRC16 [Hopps (2000)].

Existing VLB implementation, which randomly picks a core as the intermediate hop, behaves equivalently to ECMP [Al-Fares et al. (2010)] and can lead to collisions at the cores. However, if an end host periodically picks a random core as the intermediate hop, there will be no permanent collision. As a result, we implement this *periodical VLB* (*pVLB*) in the testbed to compare with DARD. An end host uses Linux IP-in-IP tunneling to determine which core is the intermediate hop.

2.5.2 Simulator

To evaluate DARD in larger scale, we build a DARD simulator in *ns-2*, which captures the system's packet level behavior. This simulator supports fattree, Clos network [Greenberg et al. (2009)] and the Cisco 3-tier topology whose oversubscription is larger than one [Cisco Systems, Inc. (2011)]. TCP New Reno is used as the transport protocol.

We also implement the demand-estimation and simulated annealing algorithm described in Hedera, a centralized elephant flow allocation approach, and set the control parameters according to [Al-Fares et al. (2010)]. To fit the simulator in memory and to speed up the simulation, we remove all the unnecessary packet headers and disable almost all the tracing and logging functions. We use the flow allocation of Hedera's simulated annealing to approximate the optimal solution.

2.6 Evaluation

This section describes DARD's evaluation on both testbed and *ns-2* simulator. We focus this evaluation on three aspects. (1) DARD does not cause path oscillation and can fastly

converge to a state where every host pair stops reallocating flows. (2) DARD achieves higher bisection bandwidth than ECMP and pVLB and its gap to the optimal flow allocation is small. (3) DARD introduces less control overhead than centralized flow allocation and its overhead is bounded by the size of the topology. We conduct a fair comparison of DARD with the industry practice (ECMP) and the solutions that are incrementally deployable (pVLB and Hedera). We leave comparing DARD with MPTCP to our future work.

2.6.1 Evaluation Settings

Traffic Patterns

Due to the absence of real datacenter traffic traces, we use the three traffic patterns introduced in [Al-Fares et al. (2008)] for both our testbed and simulation evaluations. (1) ***Stride***, where an end host E_{ij} sends elephant flows to the end host E_{kj}, $i \neq k$. This pattern emulates the extreme case where traffic stresses out the links between the cores and the aggregation switches. (2) ***Staggered(*P_{ToR}*,*P_{Pod}*)***, where an end host sends flows to another end host connecting to the same ToR with the probability P_{ToR}, to any other end host in the same pod with the probability P_{Pod} and to the end hosts in different pods with the probability $1 - P_{ToR} - P_{Pod}$. In our evaluation, P_{ToR} is 0.5 and P_{Pod} is 0.3. This pattern emulates the case where the instances of the same application are close to each other and the most traffic is within the same pod or even under the same ToR switch. (3) ***Random***, where an end host sends elephant flows to any other end host with a uniform probability. This traffic pattern emulates an average case where applications are randomly placed in datacenters. The above three traffic patterns can be either $static$ or $dynamic$. The static traffic means all permanent elephant flows start at the same time. The dynamic traffic refers to the elephant flows that start to transfer large files at different times.

Parameter Settings

To ensure that the evaluation setting reflects real world datacenters as closely as possible, we carefully choose every parameter in both the testbed and simulation evaluation. This section enumerates all these parameters and explains how their values are assigned.

Depending on the datacenter designs, the network oversubscriptions vary a lot. The oversubscription ratio is computed as a layer's aggregated downstream bandwidth divided by that layer's aggregated upstream bandwidth. It is usually larger than one because the higher the hierarchy is, the more expensive the device is. On the aggregation level, the fattree and Clos network have 1-oversubscription, while the Cisco 3-tier topology has this ratio between 1.5 and 3.6. The inter-switch link capacity of all the three topologies is in the order of $10Gbps$. Due to hardware constraints, including Deterlab's maximum $100Mbps$ bandwidth and *ns-2* server's memory limitation, we scale down the above **link capacities** to $100Mbps$ in testbed and $1Gbps$ in simulation but keep **oversubscriptions** the same. Given the end-to-end latency in datacenters is in the order of μs, we set a link's one direction **latency** to $10\mu m$. The switch **buffer size** is configured as the delay bandwidth product.

We set the **elephant flow threshold** to be $100KB$, because around 20% of the production datacenter flows are larger than $100KB$ [Kandula et al. (2009b)]. For the static traffic pattern, **the number of elephants between a source and destination** is uniformly distributed between M and $2M$, where M is the number of paths connecting the pair. We choose this value to stretch the datacenter capacity and to ensure that every path connecting a pair has the chance to be allocated with elephant flows. In both testbed and simulation experiments, we set the **dynamic traffic pattern's flow size** to be $128Mb$ for two reasons. First, elephant size is usually between $100Mb$ and $1Gb$ [Kandula et al. (2009b)]. We choose the size toward the smaller value to accelerate experiments. Second, we need to ensure a flow finishing earlier is purely caused by a better flow allocation but not because

of a smaller file. We will explain how we set the flow inter-arrival times on experiment bases.

The **control interval**, the interval of two consecutive selfish flow allocations of the same host pair, is set as 5 seconds plus a random time in $[0s, 5s]$. Because the majority of elephant flows last for more than $10s$ [Kandula et al. (2009b)], our control interval setting is small enough to prevent a flow from being delivered even without getting the chance to be reallocated, and on the other hand, it is big enough to limit the reallocation frequency. It also matches Hedera's $5s$ control interval [Al-Fares et al. (2010)]. To conduct a fair comparison, we also set each end host in pVLB to randomly pick a core as the intermediate hop every 10 seconds. Since an end host in DARD does not need the latest network states until the next round of selfish flow allocation, we can set the query interval, the interval between two consecutive sets of queries for switch states sent by the same host, to be $5s$ which is the lower bound of the above control interval. However, to ensure an end host receives the latest switch states, we add a safe margin and set this **query interval** to be $2.5s$, *i.e.*, at least two sets of responses are received by the end host before the next round of selfish flow allocation.

Last but not least, we intend to choose a right value for δ in the selfish flow allocation algorithm. We have proven that DARD converges to a Nash equilibrium despite the δ value in Chapter 2.4.2. We also bound the gap between DARD's and the optimal equilibriums if δ satisfies (2.5) in Chapter 2.4.3. However, we cannot obtain either side of (2.5) until the system converges. As a result, we treat δ as a variable in our evaluation and vary it from $1Mbps$ to $10Mbps$ to evaluate how sensitive is DARD to different δ values. We consider this range reasonable because the number of concurrent flows going in and out of a host is almost no more than 100 [Greenberg et al. (2009)]. Assuming a pessimistic scenario where every end host concurrently sends 100 elephants, each link in a fattree topology will approximately delivers 100 elephants. Given the $1Gbps$ link capacity in our simulation, a link's fair share is $\frac{1Gbps}{100} = 10Mbps$. We choose this value as the δ according

to (2.5). By intuition, the smaller the δ is, the larger fair share DARD will achieve when it converges. To verify this intuition we further choose two smaller δ values, $1Mbps$ and $5Mbps$, in our simulation. We only let δ be $10Mbps$ in our testbed evaluation to accelerate the experiment and to reduce the log file size on every switch and end host.

2.6.2 Testbed Results

The purpose of the testbed evaluation is to prove that DARD is readily deployable and can outperform the practical state of the art, including ECMP and pVLB, the improved version of VLB, under both static and dynamic traffic patterns.

We design the first set of testbed experiments to demonstrate a working DARD system. We inject random static traffic in the testbed. Each elephant flow is a long-lived TCP connection. We constantly log each elephant flow's rate and the time stamp when a flow gets reallocated. We obtain the bisection bandwidth by aggregating the incoming flow rates at all the end hosts.

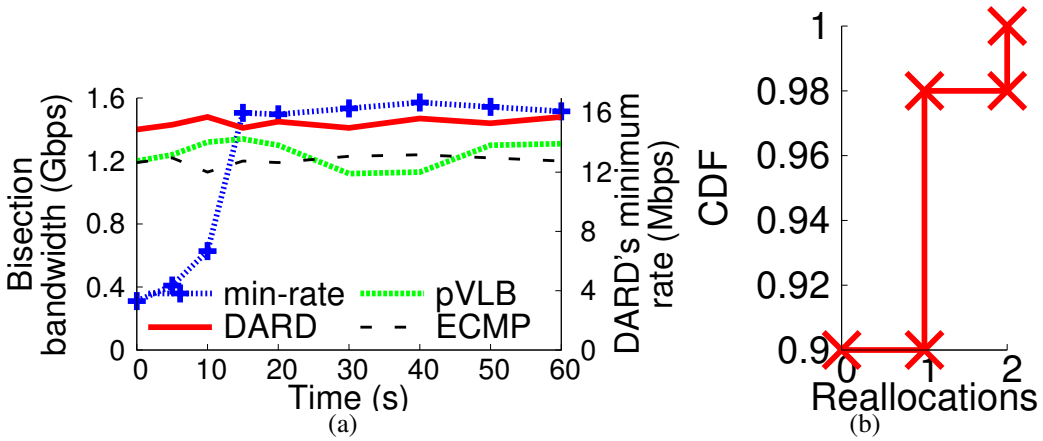


FIGURE 2.5: This figure demonstrates how DARD works from traffic injection to system convergence under random static traffic on the testbed. The dash line with cross in Figure 2.5(a) shows DARD increases the minimum flow rate over time. The other three lines in Figure 2.5(a) shows DARD achieves higher bisection bandwidth compared with ECMP and pVLB. Figure 2.5(b) shows 98% of the flows get reallocated less than once before the system converges.

As shown in Figure 2.5(a), DARD achieves 15% higher bisection bandwidth than ECMP and pVLB by reallocating flows to less congested paths. pVLB's bisection bandwidth have more fluctuation than ECMP, which implies that pVLB may just migrate a hot spot instead of eliminating hot spots. The dash line with plus shows that DARD increases the minimum flow rate by offloading flows from overloaded paths. We can also tell from Figure 2.5(a) that DARD converges to a stable state around $15s$, which is approximately 2 control intervals. Figure 2.5(b) shows each elephant gets reallocated for only limited times: 90% of the flows are not reallocated in their life-cycles, 8% are reallocated once. The remaining 2% are reallocated to less congested paths only twice.

In our second set of testbed experiments, we repeat the same measurement but under different static traffic patterns. The experiment lasts for one minute. We use the results from the middle 40 seconds to calculate the average bisection bandwidth. Figure 2.6 shows the result: DARD outperforms both ECMP and pVLB. We also observe that the bisection bandwidth gap between DARD and the other two approaches increases in the order of staggered, random and stride. It is because the flows via the core have more path diversities than the flows inside a pod. Compared with ECMP and pVLB, DARD strategically reaches a better flow allocation than simply relying on randomness.

In the third set of testbed experiments, we evaluate DARD under dynamic traffic. We stretch the capacity of the network by increasing the number of elephant flows between every source and destination and compare DARD and ECMP's flow finish times. We vary the flow generating rate of each host pair from 1 to 10 per second. The experiment lasts five minutes. We track the start and the end time of every flow and calculate the average finish times for both DARD and ECMP. We then calculate the time improvement using formula (2.12), where T_{ECMP} is the average file transfer time with ECMP, and T_{DARD} is the average file transfer time with DARD.

$$improvement = \frac{T_{ECMP} - T_{DARD}}{T_{ECMP}} \tag{2.12}$$

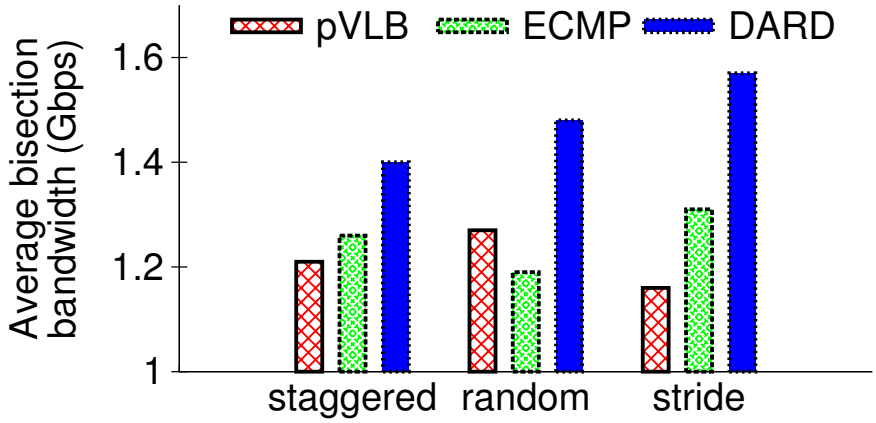


FIGURE 2.6: This figure compares DARD, ECMP and pVLB's average bisection bandwidths under different static traffic patterns on the testbed. The more path diversities, the larger bisection bandwidth DARD achieves.

Figure 2.7 shows DARD's average improvement over ECMP *vs.* the flow generating rate under different traffic patterns. For the stride pattern, DARD outperforms ECMP because it reallocates flows from overloaded paths to underloaded paths and increases the minimum flow rate in every round. We find random and staggered traffic share an interesting pattern: When the flow generating rate is low, ECMP and DARD have almost the same performance because the bandwidth is over-provided. As the flow generating rate increases, cross-pod flows congest the switch-to-switch links, where DARD can reallocate the flows off the bottleneck and improve the file transfer time. When the flow generating rate becomes even higher, the links between hosts and switches are congested by flows within the same pod and become the bottlenecks, which cannot be bypassed by any flow allocation algorithm. The comparison of DARD and pVLB follows the same pattern.

In our last set of testbed evaluations, we enhance the previous experiment by using dynamic traffic of different inter-arrival times to mimic the reality. Because the inter-arrival times of all the flows follow a periodic pattern spaced by $15ms$ and around 20% of all the flows are elephants [Kandula et al. (2009b)], we set a new elephant flow to start at an end host every $75ms$. Figure 2.8 shows DARD improves the medium file transfer time

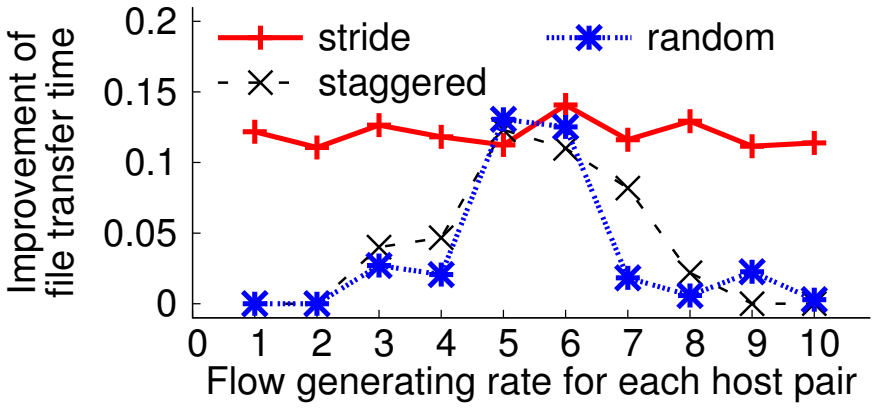


FIGURE 2.7: This figure shows DARD's improvement of file transfer time compared with ECMP under dynamic traffic patterns on the testbed.

by pushing both the maximum and minimum toward the medium. The implication is that DARD improves bisection bandwidth and fairness under real datacenter traffic patterns.

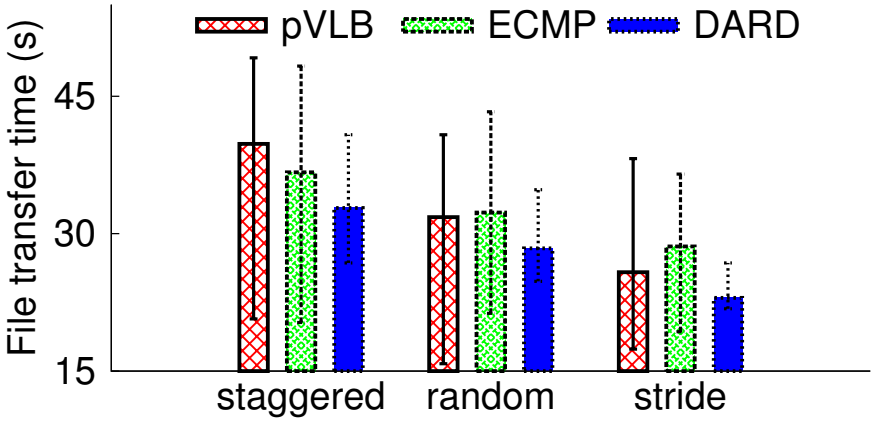


FIGURE 2.8: This figure shows ECMP, pVLB and DARD's maximum, medium and minimum file transfer times under dynamic traffic patterns. DARD reduces the file transfer time by pushing both the maximum and minimum toward the medium.

2.6.3 Simulation Results

One advantage of simulation is that we can evaluate DARD in different topologies and in much larger scale. We also compare DARD with Hedera, a centralized elephant allocation

approach, in the *ns-2* simulation.

Path Oscillation and Convergence Speed

Adaptive routing may lead to path oscillation because flows are synchronously allocated to underloaded paths [Khanna and Zinky (1989)]. Our first set of simulation is to show that DARD does not cause path oscillation and can quickly converge to a stable state, where every host pair stops reallocating flows.

We start from showing the existence of path oscillation if we do not randomize the control interval. We disable the randomization by setting all control intervals to be $5s$ and conservatively set the δ of the selfish flow allocation algorithm to be $1Mbps$. Then we inject dynamic random traffic pattern on a 128-host fattree (consisting of 8-port switches). Since the core's outgoing links are usually the bottleneck for inter-pod elephant flows [Al-Fares et al. (2010)], we track the utilizations at the core. Figure 2.9 shows the utilizations on the eight outgoing links of the first core. After the initial oscillation due to flow reallocations, the utilizations stabilize afterward. However we cannot simply conclude that DARD does not cause any path oscillation, because the above link utilizations cannot illustrate a single flow's behavior. As a result, we further log every flow's path changes and find out that even after the the utilizations stabilize, certain flows are constantly reallocated between two paths, *e.g.*, one elephant flow is reallocated between two paths 23 times in its life cycle. This indicates path oscillation exists without randomization.

We address the above path oscillation by adding a random time between $0s$ and $5s$ to the control interval. We conduct the same experiment as Figure 2.9 under dynamic traffic patterns. Figure 2.10 shows the CDF of how many times flows are reallocated in their life cycles. 86% of the staggered flows stick to their original paths because of the limited path diversities within the same pod. Almost all of the stride flows get reallocated for only less than four times even though they are across pods and have the most path diversities. These small numbers of flow reallocations indicate DARD is stable after randomization.

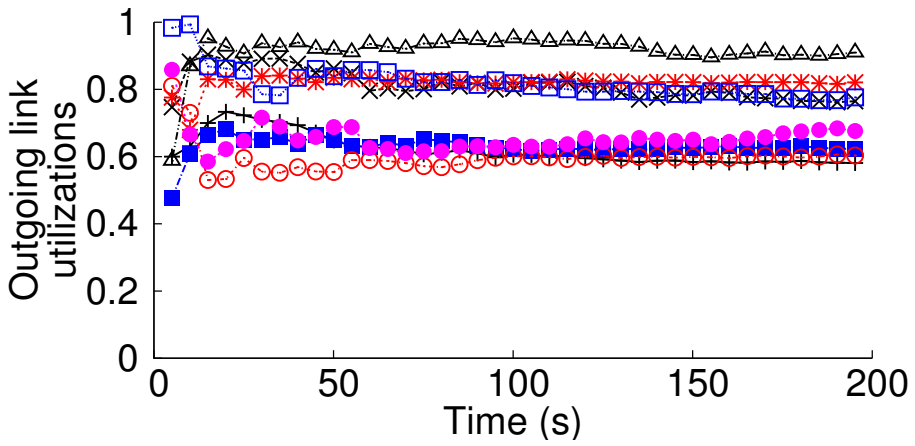


FIGURE 2.9: This figure shows the outgoing link utilizations of the first core under dynamic random traffic pattern. These utilizations stabilize after the initial oscillation even without control interval randomization. However this figure does not reflect a single flow's behavior.

We have proven (Chapter 2.4.2) and shown that DARD can converge to a Nash equilibrium. However, if the convergence takes a significant amount of time, the network will be underutilized during this process. As a result, we also measure how fast DARD can converge to the Nash equilibrium. We use the static traffic pattern on a fattree with 1024 hosts. We start all the elephant flows simultaneously and track the time each host pair stops reallocating flows. Figure 2.11 shows the CDF of the above times. DARD converges in less than $25s$ for almost all the host pairs. Given a control interval at each end host is roughly 10s, the entire system converges in less than three control intervals. DARD has the fastest convergence speed under staggered traffic, because its dominant intra-pod flows have less path diversities.

Figure 2.10 and 2.11 have shown when $\delta = 1Mbps$, DARD does not cause path oscillation and almost all of the host pairs can converge to a stable state within two to three control intervals. We also conduct the same experiments when δ is assigned with $5Mbps$ and $10Mbps$. Figure 2.12 shows the 90-percentile of the times a flow gets reallocated given different δ values. We analyze this figure in two aspects. First, given the same δ

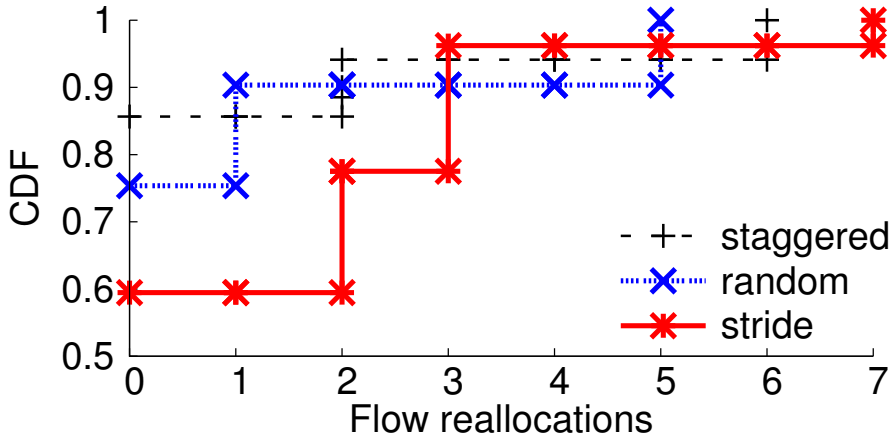


FIGURE 2.10: This figure shows the CDF of flow reallocations during their life cycles after control interval randomization. For all the dynamic traffic patterns, more than 90% of flows get reallocated less than 4 times.

value, stride traffic causes the most flow reallocations because inter-pod traffic has more path diversities. It is also consistent with the observation in Figure 2.10. Second, given the same traffic pattern, each host pair in DARD takes less steps to converge when δ increases. This is because larger δ makes DARD less sensitive to the fair share difference that is smaller than δ, in which case host pairs quickly stop reallocating flows.

Impact of Topologies, Traffic Patterns and δ Values

We prove DARD converges to a Nash equilibrium despite the δ values (Chapter 2.4.2). We also prove that if we carefully choose this δ, the gap between DARD's and the optimal flow allocation is bounded (Chapter 2.4.3). In our second set of *ns-2* evaluation, we are going to enhance the above theories by showing DARD's two advantages despite different topologies and traffic patterns. First, the bisection bandwidth achieved by DARD is close to that achieved by centralized flow allocation. Second, different δ values do not significantly impact DARD's performance.

We conduct our simulations on **three topologies**: 1024-host fattree, 1024-host Clos network [Greenberg et al. (2009)] and 1024-host 3-tier Cisco topology [Cisco Systems,

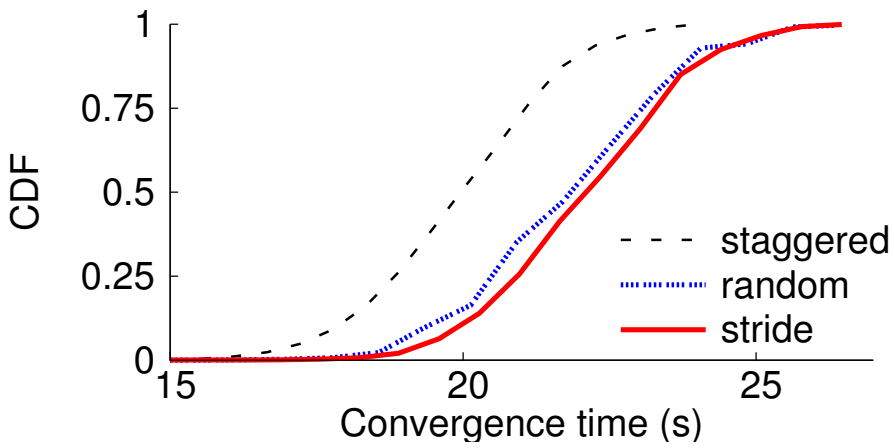


FIGURE 2.11: This figure shows the CDF *vs.* the time when a host pair stops reallocating flows. Every host pair stops reallocating flows within 3 control intervals under three static traffic patterns.

Inc. (2011)]. The fattree and Clos topologies all have 1-oversubscription, but the Cisco topology has 1.5-oversubscription on the aggregation level. We inject **three static traffic patterns** on these topologies and measure the bisection bandwidth by constantly aggregating the receiving rate of every end host. For the same topology, the same traffic pattern, we compare **six approaches** of flow allocation: Hedera, ECMP, pVLB and DARD with $\delta = 1Mbps$, $5Mbps$ and $10Mbps$. For the same setting, we conduct the experiment ten times. As a result, we conduct $3 \times 3 \times 6 \times 10 = 540$ simulations in total. Every experiment lasts for five *ns-2* minutes. We average the bisection bandwidth from the last four *ns-2* minutes for analysis. We consider Hedera as the close-to-optimal solution and use Hedera's bisection bandwidth to normalize the bisection bandwidth of the other approaches under the same topology and the same traffic pattern. As a result, Hedera's normalized bisection bandwidth is always 1. Figure 2.13 shows the normalized bisection bandwidth of different flow allocation approaches. We analyze this figure from three aspects.

First, we consider the impact of different traffic patterns under the same topology. For the fattree topology, DARD always outperforms ECMP and pVLB. The bisection bandwidth gap between DARD and these two approaches increases when inter-pod traffic

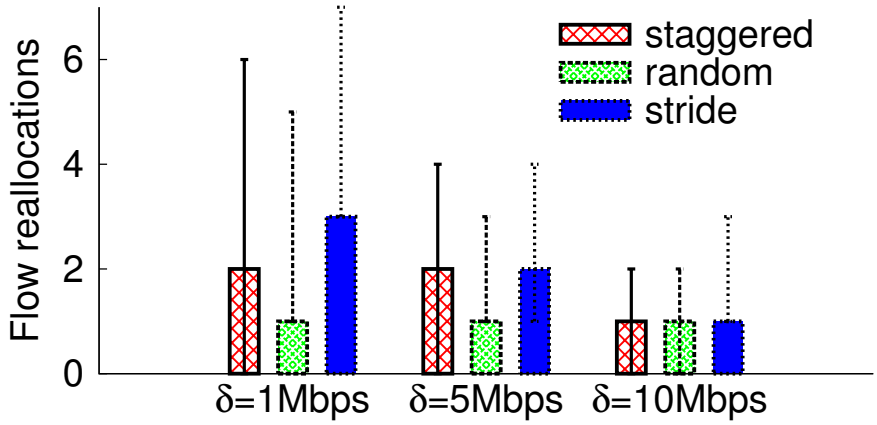


FIGURE 2.12: This figure shows the 90-percentile of the times a flow gets reallocated given different δ values and different traffic patterns. The error bar shows the 80-percentile and 100-percentile. Stride traffic introduces the most flow reallocations due to its dominant inter-pod flows. DARD converges faster when δ increases from $1Mbps$ to $10Mbps$.

becomes dominant. This result matches our testbed evaluation in Figure 2.6. Another observation is that under the staggered traffic, where intra-pod traffic is the dominant, DARD outperforms Hedera by around 20%. This is because Hedera only reallocates the flows going through the cores and it degrades to ECMP when most of the traffic is intra-pod [Al-Fares et al. (2010)]. On the other hand, under the stride traffic, where inter-pod traffic is dominant, Hedera achieves larger bisection bandwidth than DARD, but the gap between DARD and Hedera is less than 25%. We have the same observation under both the Clos network and Cisco topologies.

Second, we consider the impact of different topologies under the same traffic pattern. Both fattree and the Clos network have 1-oversubscription. Cisco topology, on the other hand, has 1.5-oversubscription on the aggregation level. This causes the Cisco topology to be more likely bottlenecked between the aggregation switches and the cores. As a result, under the stride traffic, where inter-pod traffic is the dominant, Hedera has the best performance on the Cisco topology. ECMP and pVLB reach only 30% of Hedera's bisection bandwidth. This number for DARD is between 60% and 70%. The implication

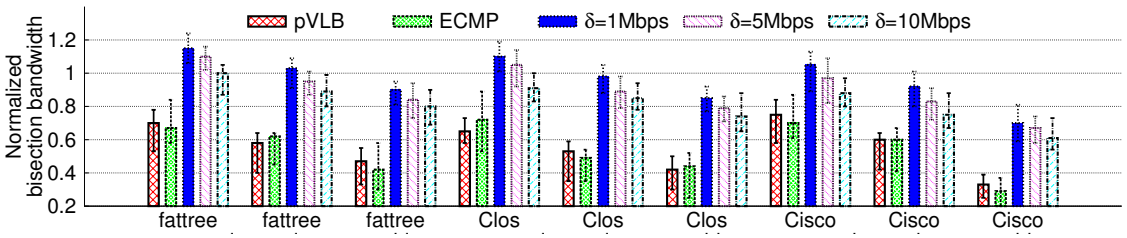


FIGURE 2.13: This figure compares the maximum, medium and minimum bisection bandwidth of different flow allocation approaches. For the same topology and traffic pattern, we use Hedera's bisection bandwidth to normalize the bisection bandwidth of the other approaches. DARD outperforms both ECMP and pVLB under all circumstances. It also outperforms Hedera when intra-pod traffic is dominant. When inter-pod traffic is dominant, DARD's gap to Hedera is small. Larger δ can slightly decrease DARD's bisection bandwidth.

is that as the state of the art industry practice, hashing-based randomness is not efficient for allocating elephant flows on oversubscribed networks.

In the last aspect, we consider the impact of different δ values. Given the same topology and the same traffic pattern, we observe that the larger δ is, the smaller DARD's bisection bandwidth is. We will explain the reason in detail in Chapter 2.4.3 and only provide the intuition here: A small δ makes a source and destination pair sensitive to the differences in flow rates and thus, forces the pair to explore better flow allocations as long as the maximum difference is beyond this δ. In this set of experiments, we increase δ by 10 times (from $1Mbps$ to $10Mbps$), the bisection bandwidth achieved by DARD decreases by at most 20%, which implies that different δ values do not significantly impact DARD's performance.

Control Overhead

In our last set of simulations, we compare DARD and Hedera's control overhead. DARD's control traffic is mainly caused by periodical polling, including both switch state queries and switch state replies. Hedera's control traffic consists of three parts: controller's queries to the ToRs for the elephants, ToRs' replies and the flow entry update messages from the

controller to all the switches. DARD's control traffic is not related to δ because it is the query interval that determines how often an end host sends out queries to switches. As a result, δ is set to $1Mbps$ for all the experiments. We trace the control traffic of both DARD and Hedera on a 128-host fattree under a static random traffic pattern. We also increase the number of elephants between all the source and destination pairs to measure how the control overhead would change with the traffic load. Figure 2.14 shows the maximum bandwidth taken by the control traffic *vs.* the peak number of elephants in the topology. We analyze this figure from two aspects.

We initially compute the bandwidth taken by the control messages by averaging all the control traffic every 30 seconds. As shown in Figure 2.14, we divide the x-axis into three stages. In the first stage, DARD's control messages take less bandwidth than Hedera's, mainly because of DARD's smaller message size (48 or 32 bytes for DARD but 80 or 72 Bytes for Hedera). In the second stage, DARD's control messages take slightly more bandwidth. That is because one new elephant flow introduces potentially more control messages in DARD than in Hedera. In an extreme example, when migrating a flow, Hedera only needs to insert entries along the new path and delete the entries along the old path. On the other hand, the source host in DARD has to query the switches along all the possible paths. In the third stage, DARD's polling traffic is eventually bounded by the topology size. However, Hedera's overhead increases proportionally to the number of elephants.

Since a large computation period may mask short-lived spikes, we conduct the same computation every 5 seconds. The result is also shown in Figure 2.14. The curve for DARD does not change much because each end host's individual behavior smooths out the spikes. On the other hand, the maximum bandwidth cost by Hedera increases significantly (as much as 3 times in the worst case). The implication is that centralized flow allocation introduces considerable temporal control overhead. If some forwarding entries are not updated simultaneously, Hedera falls back to ECMP.

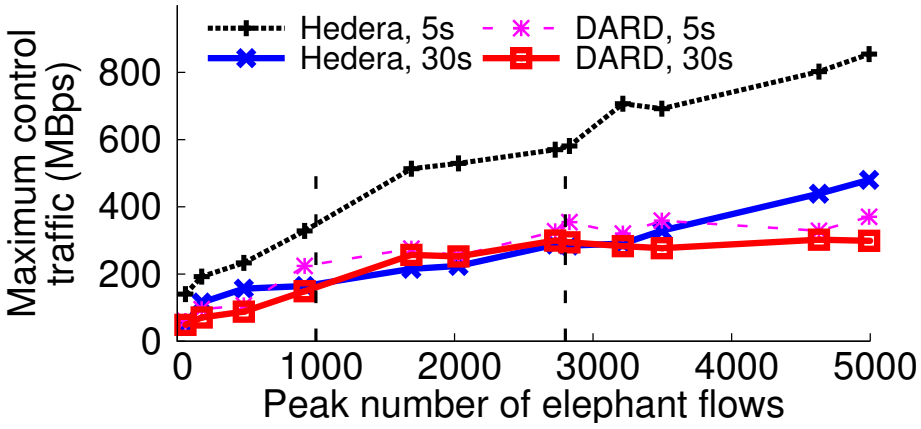


FIGURE 2.14: This figure shows the comparison of DARD and Hedera's control overhead. DARD's control traffic is bounded by the topology size and does not increase with the number of elephants. However, Hedera's control traffic is approximately proportional to the number of elephants. Hedera leads to larger control traffic when we choose a smaller computation interval, which indicates that Hedera introduces control traffic spikes.

2.7 Summary

This chapter has proposed DARD, a readily deployable distributed routing architecture for datacenter networks. It is reliable, capable of improving bisection bandwidth, scalable and debugging-friendly. DARD allows each end host to selfishly reallocate elephant flows from overloaded paths to underloaded paths. Our analysis shows that DARD converges to a Nash equilibrium in finite steps and DARD's gap to the optimal flow allocation is bounded in the order of $\frac{1}{logL}$ with L being the number of links. Testbed deployment and *ns-2* simulation indicate that the above gap is small in practice and that DARD introduces small control overhead.

3

MPXCP: Explicit Multipath Congestion Control Protocol

3.1 Introduction

In the past decade, the Internet has undergone two fundamental evolvements: on the one hand, the explosive demand for deploying data-intensive services in cloud has driven the datacenter network to upgrade to $10Gbps$ and $40Gbps$ link capacities. In addition, to fulfill the capacity demand, modern datacenters are designed to have multiple paths connecting any host pair. This design enlarges the end-to-end capacity by tens of times. On the other hand, the world-wide demand for mobile computing has connected mobile devices to the Internet seamlessly via both WIFI and LTE, even airline flights have provided wireless connections. Both evolutions lead to a consequence of increasingly larger delay-bandwidth product.

However, TCP, the dominant transport protocol, has significantly limited the capability of the underlying infrastructures. First, TCP is sensitive to packet reordering. Consequently, almost all the practical multipath routing protocols [Wu and Yang (2012); Al-Fares et al. (2010)] are allocating traffic in the granularity of a TCP connection. Without

the capability of stripping one TCP connection into multiple paths, collisions and suboptimal flow allocations may significantly harm the aggregated throughput [Al-Fares et al. (2010)]. Second, with the increase of the delay-bandwidth product in both datacenter and wireless networks, TCP becomes oscillatory and prone to instability [Katabi et al. (2002)].

To eliminate the first limitation, multipath TCP (MPTCP) [Wischik et al. (2011)] has been proposed to upgrade the transport layer to support multiple paths rather than relying on complex routing protocols. Instead of carefully allocating every single TCP connection, MPTCP views the multiple paths as a resource pool and tries to efficiently and fairly utilize it by extending TCP's AIMD concept. Intensive evaluations have shown that MPTCP can efficiently and fairly utilize network bottlenecks [Raiciu et al. (2011, 2012)].

However, to the best of our knowledge, MPTCP focuses on capacity allocation and sequence number management. It does not address TCP's limitation under increasing delay-bandwidth product. We summarize MPTCP's performance degradation due to large delay-bandwidth product into **four** aspects: **slow convergence**, **inefficiency**, **being biased against long-RTT flows** and **large queue size**. Chapter 3.2 explains them in detail. The key insight is: MPTCP and the legacy TCP share common design principle and technique details. As a result, they suffer form similar degradation when the delay-bandwidth production increases. This degradation can even be resolved by similar solutions.

In this chapter, we choose to tackle the four performance degradations from two perspectives. On the one hand, we leverage MPTCP's congestion control law to efficiently and fairly utilize the multiple paths connecting a host pair. On the other hand, we use explicit congestion feedback to accelerate the convergence [Katabi et al. (2002)]. We refer to this transport protocol as *Explicit Multipath Congestion Control Protocol* (*MPXCP*). We carefully design MPXCP such that switches do not have to maintain per-flow state, and thus MPXCP can scale up to arbitrary numbers of flows. Our intensive simulation shows that MPXCP can address all the four MPTCP's degradations despite different delay-bandwidth products.

To the best of our knowledge, MPXCP is the first explicit multipath congestion control protocol. Our contributions are:

1. We discover four MPTCP performance degradations: slow convergence, inefficiency, being biased against long-RTT flows and large queue size.

2. We design MPXCP to efficiently and fairly utilize network capacity despite its delay and bandwidth.

3. We implement MPXCP in *ns-2*. Intensive simulation shows that MPXCP outperforms MPTCP despite different delay-bandwidth products.

The rest of the chapter is organized as follows. Chapter 3.2 reviews the design rationale of MPTCP and its degradation under large delay-bandwidth product. We introduce MPXCP's design detail in Chapter 3.3. Chapter 3.5 explains MPXCP's stateless implementation. Chapter 3.6 describes our intensive *ns-2* evaluation. We also explore other design considerations extended from MPXCP's control logic in Chapter 3.7. Chapter 3.8 discusses related work and Chapter 3.9 summarizes this chapter.

3.2 MPTCP Summary

3.2.1 MPTCP's Congestion Control

Multipath TCP [Wischik et al. (2011)] considers the multiple paths connecting a host pair as a resource pool. Once the source puts the traffic of one connection along all the paths, MPTCP ensures efficiency and fairness. We use *flow* to refer to a MPTCP connection. One flow consists of multiple *sub-flows*, each of which uses a *path*.

The major complexity of MPTCP's congestion control is from the *flappiness* of *coupled congestion control* and the RTT mismatch of different paths [Wischik et al. (2011)]. To intuitively explain how MPTCP address these challenges, we assume different paths share the same RTT. We use w_s to denote the congestion window along path s and $w_f =$

$\sum_{s \in f} w_s$ to denote the aggregated congestion window of flow f. Then MPTCP extends TCP's AIMD as follows.

1. *AI*: in one RTT, increase w_s by $\frac{w_s}{w_f}$ of a packet and thus increase w_f by one packet.

2. *MD*: decrease w_s by $\frac{w_s}{2}$ after a loss on path s.

The above congestion control law puts more traffic on less congested paths and meanwhile, keeps sufficient probe traffic along congested paths [Wischik et al. (2011)].

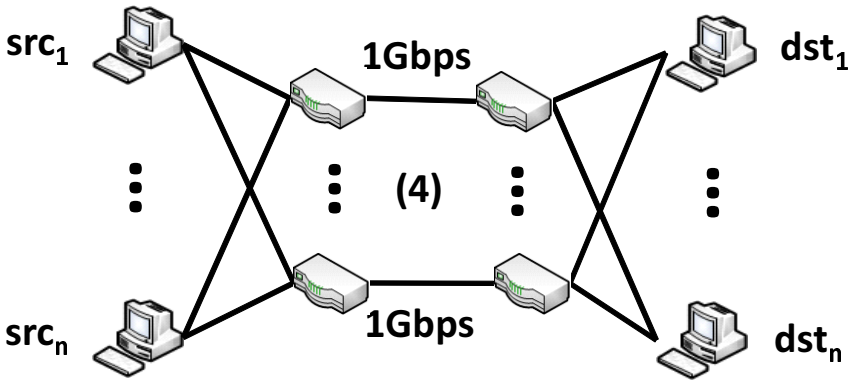


FIGURE 3.1: This figure shows the fence topology, a mimic of the TCP dumbbell topology. n host pairs are connected via four $1Gbps$ bottlenecks. Each bottleneck has a drop tail queue with the size of the delay-bandwidth product. Each host pair's RTT is experiment specific.

3.2.2 MPTCP's Degradations

If we consider the multiple paths connecting a host pair as one conceptual path, MPTCP's congestion control is exactly TCP's AIMD. As a result, we suspect that MPTCP and TCP may suffer from similar problems. To justify our concern, we use MPTCP's *ns-2* implementation [Nishida (2010)] and conduct four experiments on the fence topology shown in Figure 3.1.

In our first simulation, we use five host pairs and assign their RTTs to be $1ms$. The k^{th} host pair starts an MPTCP flow using all the four bottlenecks at the time second $20k$.

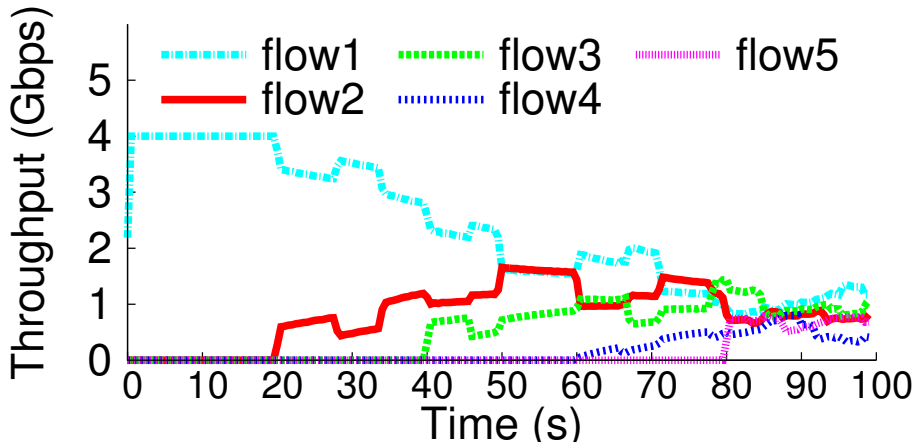


FIGURE 3.2: This figure shows MPTCP's slow convergence to fairness. Five host pairs in the fence topology start one flow every 20 seconds. Even though the bottlenecks are efficiently utilized, the bandwidth allocation is not fair.

Figure 3.2 shows each flow's throughput over time. Even though the aggregated throughput reaches the bottleneck capacity, the time to converge to fairness is long. For example, $flow_1$ and $flow_2$ do not reach fairness after $flow_2$ has been injected for 20 seconds. The negative impact of slow convergence is that the network may be inefficiently and unfairly utilized during convergence.

In our second simulation, we enlarge the bottleneck bandwidth from $1Gbps$ to $100Gbps$ but keep the same $1ms$ RTT. This setting is to mimic a datacenter network that has large bandwidth but small latency. As shown by the plus curve in Figure 3.4(a), with the increase of the capacity, the bottleneck becomes less utilized. For example, when the bottleneck increases to $100Gbps$, its utilization drops to 85%. That is because MPTCP's slow AI takes a considerable amount of RTTs to ramp up the large bandwidth.

In our third simulation, we use 30 host pairs and assign different RTTs to each pair from $1ms$ to $30ms$. The plus curve in Figure 3.8 shows that MPTCP is biased against flows with longer RTTs. We use an analytic model to explain the reason. Suppose all the paths share the same negligible loss rate p and the same RTT. In the equilibrium of a host pair, the ACK rate times the window increase per ACK should be equal to the packet drop

rate times the window decrease per drop, *i.e.*, Equation (3.1).

$$\frac{w_s(1-p)}{RTT} \times \frac{1}{w_f}\frac{w_s}{w_f} = \frac{w_s p}{RTT} \times \frac{w_s}{2} \tag{3.1}$$

Then we get $w_f = \sqrt{2(1-p)/p} \approx \sqrt{2/p}$ and $rate_f = \sqrt{2/p}/RTT$, which means each flow's rate is inversely proportional to its RTT.

In our last simulation, we let the bottleneck bandwidth be $1Gbps$ and all the RTTs be $1ms$. We start 30 MPTCP flows simultaneously. The dotted line in Figure 3.3 shows the aggregated queue length, normalized by the delay-bandwidth product, over time. Once the queue is ramped up by MPTCP, it never drains.

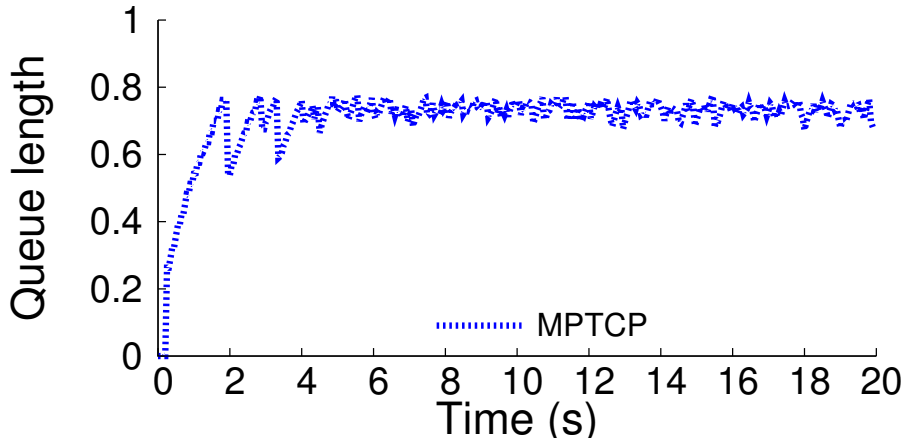


FIGURE 3.3: This figure shows the normalized queue size in the bottlenecks over time. MPTCP quickly ramps up the bottleneck queue and does not drain.

In sum, we observe that MPTCP shares the same degradations as TCP under large delay-bandwidth product, including: slow convergence, inefficiency, being biased against long-RTT flows and large queue size.

3.3 Protocol Design

3.3.1 Explore the Design Space

Ideally, we would like to find the solution to simultaneously address all MPTCP's performance degradations. In this section, we explore all the possible solutions for each

degradation. The root cause for MPTCP's slow convergence and inefficiency under large delay-bandwidth product is that the additive increase phase costs too many RTTs to ramp up the large bandwidth. A natural way to address these problems is to apply explicit congestion control [Katabi et al. (2002)], in which switches inform the end host explicitly how to set the congestion window. Like TCP, MPTCP is also biased against flows with long RTTs. There are two ways to address this problem: fair queuing [Bennett and Zhang (1996)] and explicit congestion control [Katabi et al. (2002)]. Last but not least, to maintain a small queue size, active queue management (AQM) [Floyd and Jacobson (1993)] and explicit congestion control [Katabi et al. (2002)] are two obvious choices. In sum, even though fair queuing and AQM can address part of the four MPTCP performance degradations, explicit congestion control has the potential to simultaneously address all of them. As a result, we choose explicit congestion control as our solution.

3.3.2 Explicit Congestion Control

In this section we assume a pure MPXCP network. MPXCP is a window-based congestion control protocol. It leverages both ECN and multipath to achieve efficiency and fairness. Each flow f consists of a set of *sub-flows*, each of which may take a different path. Each sub-flow s maintains its own congestion window $cwnd_s$ and its own round trip time RTT_s. A flow f maintains its aggregated congestion window according to Equation (3.2).

$$cwnd_f = \sum_{s \in f} cwnd_s. \tag{3.2}$$

The sender informs the switches with its congestion windows via a congestion header in every packet. The switch monitors the bandwidth, queue size and traffic load and tags each packet with the expected changes in each sender's congestion window. A more congested switch can further update this tag. The receiver simply echoes the tag to the sender. The sender then updates its congestion windows accordingly. MPXCP achieves the same bandwidth allocation as MPTCP but with fewer round trips. A simplified congestion window

control law is described as follows.

1. *AI*: in one RTT, increase w_s by Δw_s, where $\Delta w_s \propto w_s/w_f$ and $\sum \Delta w_s = spare\_bandwidth$.

2. *MD*: decrease w_s by Δw_s, where $\Delta w_s \propto w_s$ and $\sum \Delta w_s = queue\_size$.

In the above description, AI tries to grab the available bandwidth in one RTT and the MD tries to drain the queue in one RTT. Note that it is a highly simplified design for easy understanding. We now start to describe the design details.

3.3.3 The Congestion Header

Each MPXCP packet contains a congestion header as shown in Table 3.1. The $cwnd_s$ is the sender sub-flow s's congestion window. The RTT_s is sub-flow s's RTT estimation. The $cwnd_f$ is flow f's aggregated congestion window using Equation (3.2). These fields are initialized by the sender and never changed by the switches. The last field $feedback_s$ is initialized to be infinite by the sender. Switches on the path update this field to explicitly control the sub-flow's congestion window. A positive/negative value implies the sender needs to increase/decrease the sub-flow's congestion window accordingly.

$cwnd_s$(sender sub-flow's cwnd)
RTT_s (sender sub-flow's RTT estimation)
$cwnd_f$(sender's aggregated cwnd)
$feedback_s$(initialized to ∞)

Table 3.1: MPXCP's congestion header.

3.3.4 The MPXCP Sender and Receiver

On packet departure, the sender encapsulates the packet with a congestion header and sets RTT_s to be the corresponding round trip time estimation and $cwnd_f$ to be the flow's aggregated congestion window size. The source greedily fills infinite to the $feedback_s$ field to request window increase. When an acknowledge arrives, the sender increases the

corresponding sub-flow's congestion window according to Equation (3.3), where $size_s$ is the sub-flow's packet size.

$$cwnd_s = max(cwnd_s + feedback_s, size_s) \tag{3.3}$$

Even though packet losses caused by congestion are rare for MPXCP, packet losses due to interference are common in wireless networks. MPXCP responds to packet losses the same way as MPTCP but keeps the congestion window unchanged. In this way, packet loss is no longer an indirect signal for congestion. Congestion window is adjusted purely by the explicit congestion control.

When acknowledging a packet, the MPXCP receiver copies the congestion header from the data packet to its acknowledgement packet.

3.3.5 The MPXCP Router

A switch maintains a per link control interval estimation d. A longer control interval leads to a stagnated reaction to congestions, while a shorter control interval leads to proactive reaction to congestions even without observing the effects of the previous adjustment. As discussed in XCP [Katabi et al. (2002)], we choose the average round trip time on a link as that link's control interval.

MPXCP separates *efficiency controller* from *fairness controller*. Thus, the rule of fairly allocating capacity among flows and the rule of fully utilizing capacity are independent.

Efficiency Controller

The purpose of the *efficiency controller* is to maximize link utilization and to minimize persistent queue size. We use Equation (3.4) to refer to the aggregated feedback in a control interval d. α and β are the two constants as suggested in XCP. We prove in the appendix that XCP's α and β setting can also stabilize MPXCP despite arbitrary average RTTs. S is the spare bandwidth defined as the differences between the link capacity and

the input traffic rate. Q is the smallest queue size seen in a control interval.

$$\phi = \alpha \cdot d \cdot S - \beta \cdot Q \tag{3.4}$$

Equation (3.4) makes the feedback within one average RTT proportional to the spare bandwidth when the link is not fully utilized, or proportional to the queue size when packets are buffered. MPXCP achieves efficiency by dividing the above aggregated feedback to every packet's $feedback_s$ field.

Fairness Controller

The *fairness controller* is to allocate the above aggregated feedback to individual packets to achieve fairness. MPXCP uses MPTCP's control rationale to converge to fairness [Wischik et al. (2011)], *i.e.*, *Aggregated-Additive-Increase and Per-Subflow-Multiplicative-Decrease*. Following is how to compute per packet feedback. (1) If $\phi > 0$, a switch allocates feedback to each packet such that the throughput increases for all the flows are the same. (2) Meanwhile, the above increase of each flow is distributed among the corresponding sub-flows proportionally to each sub-flow's throughput. The intuition is that MPXCP always prefers less congested paths and also keeps sufficient probe traffic on more congested paths such that the sender can quickly react to congestion variations. (3) If $\phi < 0$, a switch allocates feedback to each packet such that the decrease in the throughput of a sub-flow is proportional to its current throughput. This feedback allocation strategy ensures MPXCP converges to MPTCP fairness when ϕ is not zero.

However, when efficiency becomes optimal, *i.e.*, $\phi = 0$, the above convergence process stops. As a result, we also introduce bandwidth shuffling as XCP [Katabi et al. (2002)] to ensure that in every control interval, at least a minimum amount of bandwidth is always reallocated. This approach ensures that the efficiency is not affected while the throughput of different flows converges to fairness. Equation (3.5) shows how to compute the shuffle traffic on each link, in which y is the link's input traffic in a control interval d. γ is set to

0.1 as XCP, which is a tradeoff between convergence speed and the fluctuation around the optimal efficiency.

$$h = max(0, \gamma y - |\phi|) \tag{3.5}$$

Per packet feedback consists of a positive feedback p and a negative feedback n, as shown in Equation (3.6), where the index s represents the a sub-flow.

$$feedback_s = p_s - n_s \tag{3.6}$$

Equation (3.7) represents how to compute per packet positive feedback p_s. Sub-flow s belongs to flow f. Symbol Δ represents the increase/decrease of a value. C_p is a positive constant. This allocation ensures (1) all flows increase the same amount of throughput in each control interval and (2) a flow's aggregated positive feedback is allocated to each sub-flows proportionally to their throughput. By doing so, each sender allocates most of its traffic to less congested paths while keeping sufficient probe traffic on other paths.

$$\begin{aligned} p_s &= \frac{\Delta cwnd_s}{\text{\# of packets}} \\ &= (\Delta tput_s \cdot RTT_s)/(\frac{cwnd_s}{size_s \cdot RTT_s} \cdot d) \\ &= \Delta tput_s \cdot \frac{size_s \cdot RTT_s^2}{cwnd_s \cdot d} \\ &= (C_p \cdot \frac{cwnd_s}{cwnd_f}) \cdot \frac{size_s \cdot RTT_s^2}{cwnd_s \cdot d} \\ &= C_p \cdot \frac{size_s \cdot RTT_s^2}{cwnd_f \cdot d} \end{aligned} \tag{3.7}$$

Because the link throughput increased in one control interval is $\frac{h+max(\phi,0)}{d}$, we have

$$\sum_{\text{all pkts}} \frac{p_s}{RTT_s} = \frac{h + max(\phi, 0)}{d} \tag{3.8}$$

Plug (3.7) into (3.8), we get

$$C_p = (h + max(\phi, 0))/\sum \frac{size_s \cdot RTT_s}{cwnd_f} \tag{3.9}$$

The per packet negative feedback is proportional to the corresponding sub-flow's throughput, as shown in Equation (3.10), where C_n is a negative constant.

$$\begin{aligned} n_s &= \frac{\Delta cwnd_s}{\text{\# of packets}} \\ &= (\Delta tput_s \cdot RTT_s)/(\frac{cwnd_s}{size_s \cdot RTT_s} \cdot d) \\ &= \Delta tput_s \cdot \frac{size_s \cdot RTT_s^2}{cwnd_s \cdot d} \\ &= (C_n \cdot cwnd_s) \cdot \frac{size_s \cdot RTT_s^2}{cwnd_s \cdot d} \\ &= C_n \cdot \frac{size_s \cdot RTT_s^2}{d} \end{aligned} \tag{3.10}$$

Because the link throughput decreased in one control interval is $\frac{h+max(-\phi,0)}{d}$, we have

$$\sum_{\text{all pkts}} \frac{n_s}{RTT_s} = \frac{h + max(-\phi, 0)}{d} \tag{3.11}$$

Plug (3.10) into (3.11), we get

$$C_n = (h + max(\phi, 0))/\sum size_s \cdot RTT_s \tag{3.12}$$

Each router will update C_p and C_n at the end of every control interval. On every packet departure, the router will compute and update the per packet feedback according to (3.7) and (3.10) using the latest C_p and C_n. One thing to note is that even $cwnd_s$ is not directly used in the above computation, though we still keep it in the congestion header to compute per sub-flow's average RTT (chapter 3.5).

3.4 Stability Analysis

This proof is similar to Katabi et al. (2002). We claim no credit for it. Consider a fence topology as shown in Figure 3.1, in which n MPXCP host pairs are sending flows via m paths. Let d be the common RTT of all host pairs. We use $y(t)$ to denote the aggregated throughput along one bottleneck at time t and $q(t)$ to denote that bottleneck's queue size at time t. According to Equation (3.4), we use the following differential equations to model the bottleneck congestion control.

$$\begin{aligned} q'(t) &= y(t) - c \\ y'(t) &= \frac{1}{d^2}(\alpha \cdot d \cdot (c - y(t-d)) - \beta \cdot q(t-d)) \end{aligned}$$

We define $x(t) = y(t) - c$, $K_1 = \frac{\alpha}{d}$ and $K_2 = \frac{\beta}{d^2}$. The above differential equations can be transformed to:

$$\begin{aligned} q'(t) &= x(t) \\ x'(t) &= -K_1 \cdot (y(t-d) - c) - K_2 \cdot q(t-d) \end{aligned}$$

As a result, the open loop transfer function is:

$$G(s) = \frac{K_1 s + K_2}{s^2} e^{-ds} \tag{3.13}$$

The magnitude and angle are:

$$|G(s)| = \frac{\sqrt{K_1^2 \omega^2 + K_2^2}}{\omega^2} \tag{3.14}$$

$$\angle G = -\pi + arctan\frac{K_1 \omega}{K_2} - \omega d \tag{3.15}$$

We follow the same logic as Katabi et al. (2002) to simplify the parameter setting, *i.e.*, setting both break frequency and crossover frequency to be $\frac{K_2}{K_1}$. Thus, when plugging $w = \frac{K_2}{K_1}$ into Equation (3.14), $|G(s)| = 1$; we get $\beta = \sqrt{2}\alpha^2$.

To ensure a stable close loop system, we need $\angle G(w_c) = -\pi + \frac{\pi}{4} - \frac{\beta}{\alpha} > -\pi$. As a result $\frac{\beta}{\alpha} < \frac{\pi}{4}$. Combined with the result from last paragraph, we have $\alpha < \frac{\pi}{4\sqrt{2}}$ and $\beta = \sqrt{2}\alpha^2$. Under this setting, the close loop system is stable despite the delay, capacity and the number of host pairs.

3.5 Implementation

We implement MPXCP in *ns-2*. The sequence number management is based on MPTCP's *ns-2* repository [Nishida (2010)]. Different from MPTCP, MPXCP's congestion control logic on the host side is trivial. Thus, we focus on the router implementation. The highlight is that routers do not have to maintain per flow state to achieve efficiency and fairness.

A MPXCP enabled router requires three pieces of code to update per packet feedback. Algorithm 2 is executed when a packet arrives. The router reads every congestion header and accumulates certain intermediate variables for the next computation of C_p and C_n.

Algorithm 3 is executed at the end of every control interval (avg_rtt). A router updates C_p and C_n and resets the timer. The aggregated positive and negative feedbacks for the next control interval are stored in $residue\_pos\_fbk$ and $residue\_neg\_fbk$.

Algorithm 4 is executed at every packet departure to update the $feedback_s$ field. The $residue\_pos\_fbk$ and $residue\_neg\_fbk$ ensure that the aggregated positive and negative feedbacks are bounded in this control interval.

Algorithm 2: On packet arrival

1: $sum\_size\_rtt\_by\_cwndf$ += $(size_s \times rtt_s)/cwnd_f$;
2: sum_size_rtt += $size_s \times rtt_s$;
3: sum_size += $size_s$;
4: $sum\_cwnds\_by\_rtt$ += $cwnd_s/rtt_s$;
5: sum_cwnds += $cwnd_s$;

Algorithm 3: On control interval timeout

avg_rtt = $sum\_cwnds / sum\_cwnds\_by\_rtt$;
ϕ = $\alpha(avg\_rtt \times capacity - sum\_size) - \beta Queue$;
$shuffle$ = max(0, $0.1 \times sum\_size - |\phi|$);
$residue\_pos\_fbk$ = $shuffle + \max(\phi, 0)$;
$residue\_neg\_fbk$ = $shuffle + \max(-\phi, 0)$;
C_p = $residue\_pos\_fbk / sum\_size\_rtt\_by\_cwndf$;
C_n = $residue\_neg\_fbk / sum\_size\_rtt$;
$sum\_size\_rtt\_by\_cwndf$ = 0;
sum_size_rtt = 0;
sum_size = 0;
$sum\_cwnds\_by\_rtt$ = 0;
sum_cwnds = 0;

Algorithm 4: On packet departure

p_s= $(C_p \times size_s \times rtt^2) / (cwnd_f \times avg\_rtt)$;
n_s= $(C_n \times size_s \times rtt^2) / avg\_rtt$;
$feedback$ = $p_s - n_s$;
if $feedback_s \geqslant feedback$ **then**
    $feedback_s$ = $feedback$;
    $residue\_pos\_fbk$ -= p_s;
    $residue\_neg\_fbk$ -= n_s;
else if $feedback_s \geqslant 0$ or $feedback \geqslant 0$ **then**
    $residue\_pos\_fbk$ -= $feedback_s$;
    $residue\_neg\_fbk$ -= $(feedback - feedback_s)$;
else
    $residue\_neg\_fbk$ += $feedback_s$;
end if
$C_p = (residue\_pos\_fbk \leqslant 0)$? 0 : C_p;
$C_n = (residue\_neg\_fbk \leqslant 0)$? 0 : C_n;

3.6 Evaluation

In this section, we use intensive *ns-2* simulation to show that MPXCP can simultaneously address all the four MPTCP's degradations discussed in Chapter 3.2. Our MPXCP's implementation is built on top of MPTCP's *ns-2* implementation from the IETF Multipath TCP working group [Nishida (2010)]. We keep the sequence number management mechanism and modify the congestion control logic as described in Chapter 3.5.

3.6.1 Simulation Setup

We use the topology shown in Figure 3.1 for most of the simulations. We vary the number of bottlenecks, number of host pairs, bottleneck capacities and host pair RTTs in a series

of experiments to compare MPXCP and MPTCP. We use drop tail queue and set the queue length to be the delay-bandwidth product for all the experiments. We set $\alpha = 0.4$ and $\beta = 0.226$ according to [Katabi et al. (2002)].

3.6.2 Higher Utilization and Smaller Queue

In this section, we will show that MPXCP achieves higher utilization and smaller queue size despite the link capacity, RTT, and the number of concurrent flows.

Impact of Capacity. We first enlarge the bottleneck capacity from $1Gbps$ to $100Gbps$ but keep the $1ms$ round trip time. This setting is to mimic a datacenter network that has large capacity but small latency. We inject 50 flows in both directions of the fence topology to stress the backward capacity, such that the ACKs can be lost. We compute the bottleneck utilizations in the forward direction. As shown in Figure 3.4(a), MPTCP cannot efficiently utilize the bottleneck when the bottleneck capacity increases. The reason is because MPTCP's additive increase takes a significant amount of round trips to ramp up the bottleneck. However, MPXCP's bottleneck utilization is close to 100%. The y-axis of Figure 3.4(b) is the queue size normalized by the delay-bandwidth product. It shows that MPXCP maintains a negligible queue while MPTCP keeps a queue size of more than 40% of the delay-bandwidth product.

Impact of Latency. We then keep the $1Gbps$ bottleneck capacity but vary the end-to-end latency from $6ms$ to $600ms$. Figure 3.5 shows the result. MPXCP can efficiently utilize the bottleneck capacity while MPTCP's utilization decreases mainly because a packet loss can easily cause timeout. Figure 3.5(b) shows that MPXCP maintains a negligible queue size. Another notable observation is that MPTCP's frequent timeout drains the bottleneck queue.

Impact of Number of Flows. We also stretch MPXCP by increasing the number of concurrent flows. To accelerate the simulation process, we set each bottleneck link to be $100Mbps$. Each host pair's round trip time is $6ms$. Figure 3.6(a) shows that both MPTCP

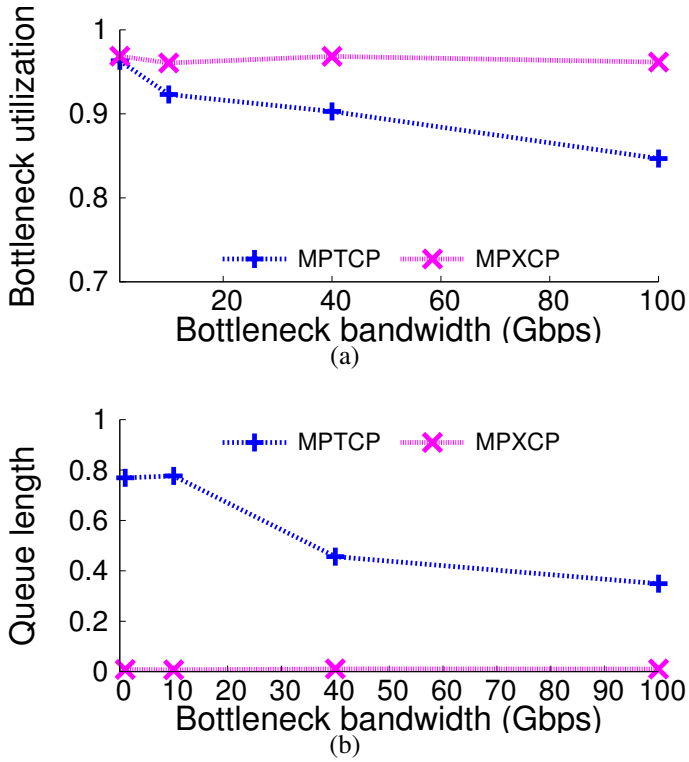


FIGURE 3.4: This figure shows the medium bottleneck utilization and queue size *vs.* the bottleneck capacity. MPXCP efficiently utilizes the bottleneck with negligible queue size despite the bottleneck capacity.

and MPXCP can efficiently utilize the bottleneck despite the number of concurrent flows. However, Figure 3.6(b) shows that only MPXCP maintains a negligible queue size. One thing to note is that when the number of concurrent flows reaches 100, *i.e.*, each MPXCP sender is stressed by maintaining a 3-packet aggregated congestion window along 4 paths, MPXCP can still efficiently utilize the bottleneck and keeps a negligible queue size.

3.6.3 MPXCP Improves Fairness

In this section, we are going to show MPXCP improves fairness despite different round trip times.

Fairness of Same-RTT Flows. We first set each bottleneck capacity to be $1Gbps$ and set all the RTTs to be $1ms$. We increase the concurrent number of flows from 1 to 30. For each number, we conduct the experiment for 200 seconds and use the flow rate from the

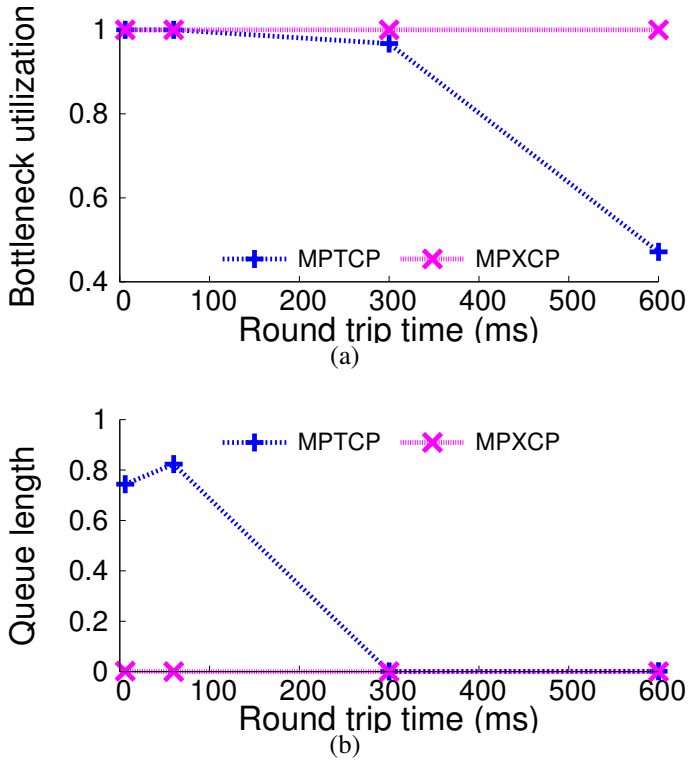


FIGURE 3.5: This figure shows the medium bottleneck utilization and queue length *vs.* RTT. MPXCP efficiently utilizes the bottleneck with negligible queue size despite the round trip time.

middle $180s$ to compute the Jain's fairness index [Chiu and Jain (1989)]. Figure 3.7 shows the medium fairness index *vs.* the number of concurrent flows. The error bars represent the maximum and the minimum. All MPXCP's mediums and error bars overlaps at value 1, while MPTCP's index varies significantly. This result implies that MPXCP is fair to the same-RTT flows despite their concurrent number.

Fairness of Different-RTT Flows. We then vary each flow's RTT according to its index: the k^{th} flow's RTT is k milliseconds. We inject 30 flows in total. Figure 3.8 shows that MPTCP is biased against long-RTT flows. Contrarily, MPXCP can evenly allocate the same amount of capacity to all the flows. The reason is because MPXCP allocates traffic in the granularity of a flow, *i.e.*, $\frac{cwnd}{RTT}$.

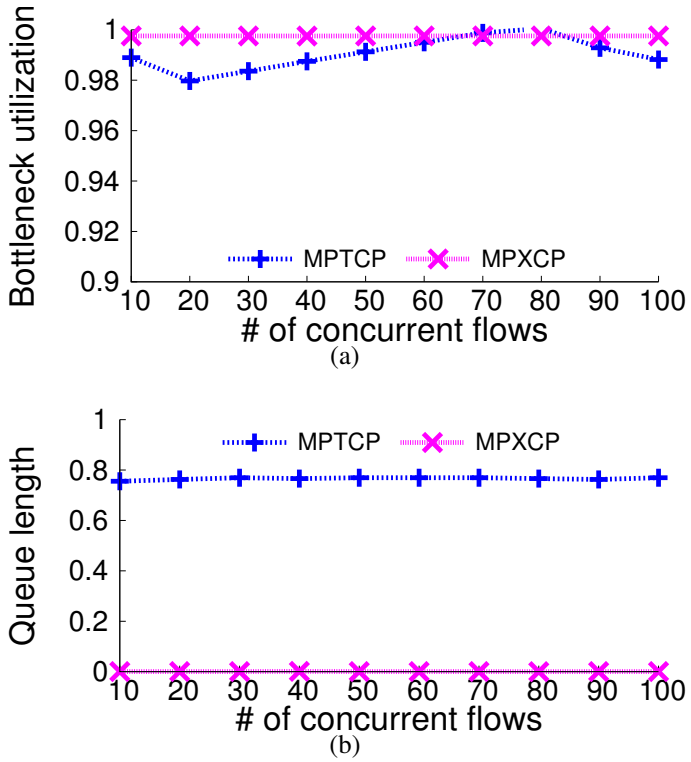


FIGURE 3.6: This figure shows the medium bottleneck utilization and queue length *vs.* the number of concurrent flows. MPXCP efficiently utilizes the bottleneck capacity with negligible queue length.

3.6.4 MPXCP Improves Convergence Speed

In this section, we will illustrate MPXCP's fast convergence.

Quick Convergence to Fairness. We first conduct the same experiment as Figure 3.2. Figure 3.9 shows that every time when a new flow is injected, MPXCP efficiently and fairly reallocates the bottleneck capacity in one round trip.

Robustness to Bursty Traffic. We also evaluate MPXCP's convergence speed under bursty traffic. We inject one long-lived flow in the fence topology at time $0s$. At time $4s$ and $8s$ we suddenly start and stop 29 other flows. Figure 3.10 shows how the bottleneck utilization and Jain's fairness index change over time. Both MPXCP and MPTCP can efficiently utilize the bottleneck capacity, but only MPXCP achieves fairness.

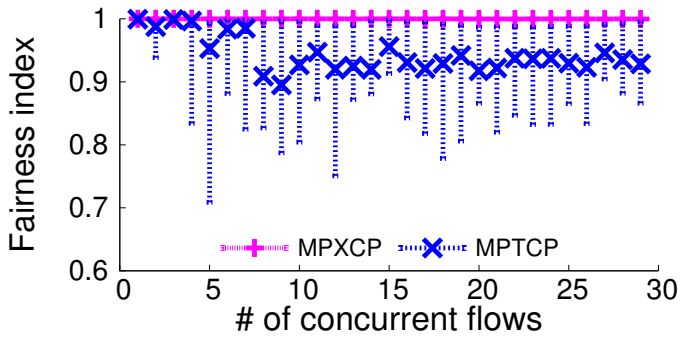


FIGURE 3.7: This figure shows Jain's fairness index *vs.* number of concurrent flows. All host pairs have $1ms$ RTT and all start at time 0s. The fairness index is computed every 0.5 second from $10s$ to $100s$. Each plus/cross represents the medium. The error bars represent the maximum and minimum. MPXCP reaches the perfect fairness.

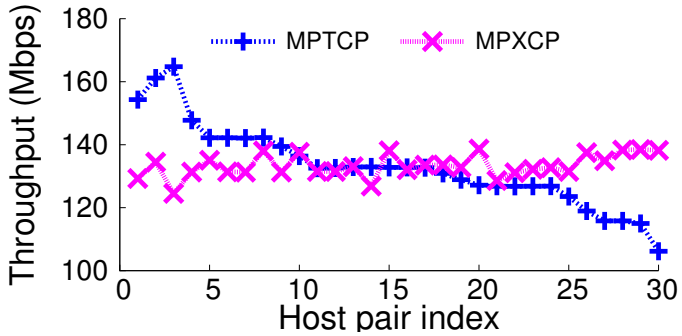


FIGURE 3.8: This figure shows each flow's throughput *vs.* its RTT. The k^{th} host pair's RTT is $k$ms. There are 30 host pairs in total. MPXCP is fair to all flows despite different RTTs because MPXCP allocates the bottleneck capacity in the granularity of a flow.

3.6.5 MPXCP Prevents Incast

In this section, we will show that because MPTCP still builds up queues, incast [Chen et al. (2009)] becomes inevitable. However, MPXCP can constantly drain the queue and prevent incast.

Figure 3.11(a) shows a typical topology under a top-of-rack switch in datacenters. The bottleneck capacity is $1Gbps$. The end-to-end RTT is $1ms$. We use the same workload described in [Chen et al. (2009)] to conduct this experiment, *i.e.*, the client requests 100 256-byte blocks of data from n servers. We vary this n from 1 to 16. Each block is striped across n servers. The client does not request block $k + 1$ until all the fragments of block

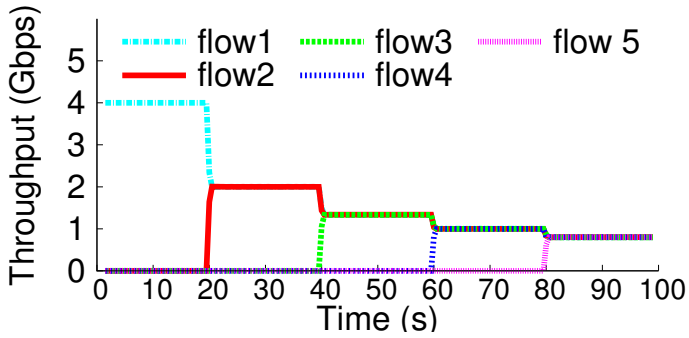


FIGURE 3.9: This figure has the same setting as Figure 3.2 but using MPXCP. It quickly converges to fairness in one RTT.

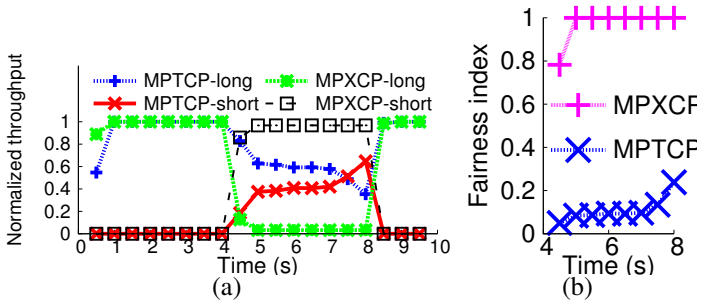
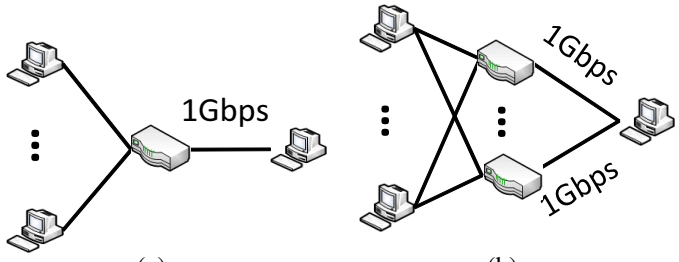


FIGURE 3.10: This figure shows that MPXCP is more robust to bursty traffic than MPTCP. One long-lived MPXCP flow is fully utilizing the bottlenecks in the fence topology. At $4s$, 29 short-lived flows are injected and the long-lived flow's throughput decreases to $\frac{1}{30}$. At $8s$, the 29 short-lived flows stop and the long-lived flow quickly grabs all the available capacity. Figure 3.10(a) shows the long-lived flow's throughput and the aggregated throughput of the 29 short-lived flows. Figure 3.10(b) shows the Jain's fairness index of all the 30 flows from $4s$ to $8s$.

k have been received. This workload results in a constantly synchronized read pattern, builds up the bottleneck queue and causes packet losses.

Figure 3.12(a) shows the aggregated goodput *vs.* the number of servers given either MPTCP or MPXCP as the transport protocol. The goodput is measured at the client by dividing the sum of received unique packets over the transmission time. Because MPTCP degrades to TCP in this topology, goodput collapse still happens when there are more than 2 servers. However, MPXCP does not cause incast due to its negligible queue size.

We further expand the topology in Figure 3.11(a) to the multipath topology in Fig-



(a) (b)

FIGURE 3.11: This figure shows the incast experiment settings. Figure 3.11(a) represents the topology under a top-of-rack switch. Figure 3.11(b) represents multihomed servers under multiple top-of-rack switches [Raiciu et al. (2011)]. In both topologies, one receiver requests data blocks from multiple senders. All RTTs are 1ms.

ure 3.11(b). This is also the topology proposed by the MPTCP community to improve MPTCP's performance [Raiciu et al. (2011)]. In this topology, each host is multihomed to m top-of-rack switches. In our evaluation, we assign m with either 2 or 4 (as shown in Figure 3.12(b)'s legend). MPTCP postpones the goodput collapse simply due to a larger aggregated bottleneck capacity. On the contrary, MPXCP prevents incast by constantly draining the queue.

3.6.6 Resilience to Network Dynamics

In this section, we will show that MPXCP is resilient to network dynamics, including packet losses and link failures [Wu et al. (2012)].

Resilience to Packet Loss. We inject 50 long-lived flows in the fence topology and increase the packet loss rate on the bottlenecks from 0 to 1%. Figure 3.13 shows the bottleneck utilization *vs.* the loss rate. MPXCP can still efficiently utilize the bottleneck capacity when the loss rate is 1%. This is because packet losses due to non-congestion reasons do not reduce the sender's congestion window.

Resilience to Link Failures. We then show that NetPilot can quickly react to link failures and recoveries. We assume an end host can seamlessly detect path availability and retransmit lost packets along a different path. How to fulfill these two assumptions is

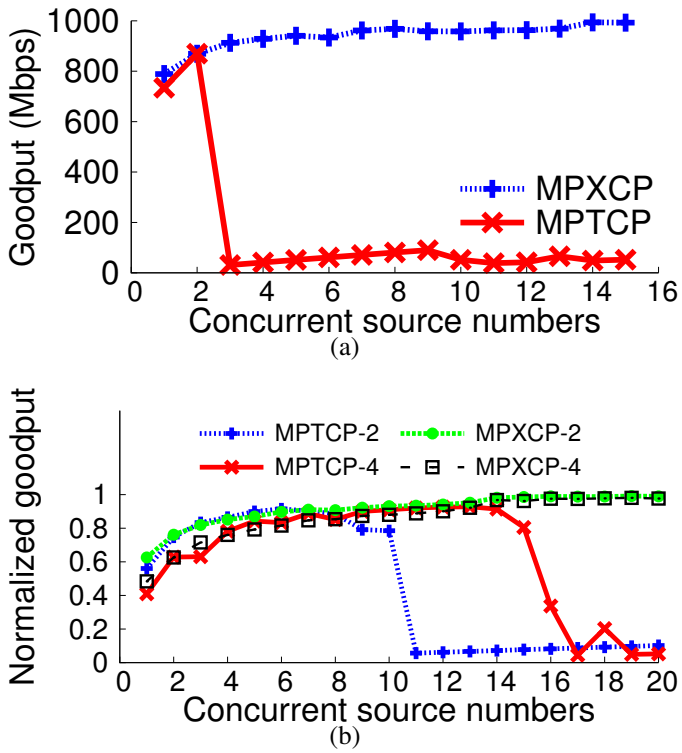


FIGURE 3.12: This figure shows how MPXCP prevents Incast. We follow the same setting as the distributed storage system [Chen et al. (2009)]. Figure 3.12(a) shows that the goodput achieved by MPXCP is almost the same as the bottleneck capacity. Figure 3.12(b) shows that if we expand the original topology (Figure 3.11(a)) into its multipath version (Figure 3.11(b)), MPTCP postpones the goodput collapse due to larger aggregated bottleneck capacity. On the contrary, MPXCP's goodput is close to the bottleneck capacity.

beyond this thesis. We inject four continuous flows in the fence topology and disconnect three bottleneck links at $2s$, $4s$, and $6s$ respectively. At $8s$, $10s$, and $12s$ we reconnect the failed links one by one. Figure 3.14 shows each flow's throughput *vs.* time. As we can see, MPXCP can quickly achieve efficiency and fairness after link failures and recoveries.

3.6.7 Sensitivity Analysis

In this section, we will analyze if MPXCP is sensitive to configurable parameters. As discussed in Chapter 3.3, MPXCP has three parameters: α and β from Equation (3.4) and γ from Equation (3.5). Given the first two parameters are computed according to control theory (Appendix), we only explore how sensitive is MPXCP to different γs. γ ensures

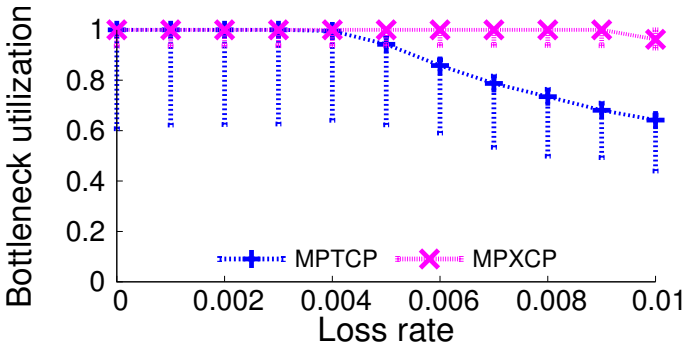


FIGURE 3.13: This figure shows MPXCP is more robust to packet losses than MPTCP. We inject 50 long-lived flows in the fence topology and increase the bottleneck loss rate from 0 to 1%. The plus/cross represents the medium bottleneck utilization from time $1s$ to $20s$. The error bars represent the minimum and maximum. MPXCP achieves larger throughput because non-congestion packet losses do not reduce the sender's congestion window.

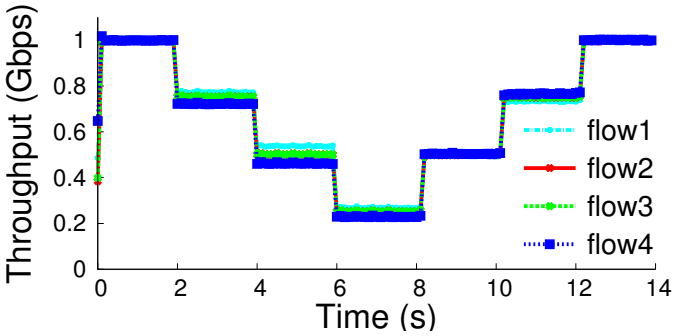


FIGURE 3.14: This figure shows MPXCP is robust to link failures. Four flows are traversing the fence topology. Three of the bottleneck links fail at $2s$, $4s$, and $6s$ respectively. The four flows always efficiently and fairly share the remaining bottlenecks. The failed bottlenecks are recovered at $8s$, $10s$, and $12s$ respectively. MPXCP share the recovered bottlenecks quickly and fairly.

that MPXCP's convergence to fairness does not stall, *i.e.*, when Equation (3.4) is close to 0 (the bottlenecks are efficiently utilized), Certain amount of capacity can still be reallocated among different flows. We start two long-lived flows in the fence topology at $0s$ and $1s$ respectively and measure the Jain's fairness index of the two flows between $1s$ and $9s$. Figure 3.15 shows that as long as γ is larger than 0, MPXCP can converge to efficiency and fairness.

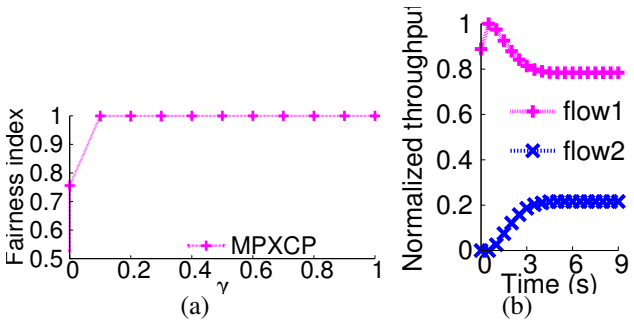


FIGURE 3.15: This figure shows that shuffle in Equation (3.5) is necessary for fairness. However, MPXCP is not sensitive to this value. We start two long-lived flows in the fence topology at $0s$ and $1s$ respectively and vary the shuffle parameter γ from 0 to 1. Figure 3.15(a) shows the medium, minimum and maximum of Jain's fairness index of the two flows between $1s$ and $10s$. Figure 3.15(b) shows that when $\gamma = 0$, the two flows can not reach fairness. Both figures indicate that MPXCP can achieve fairness as long as the shuffle percentage is positive.

3.7 QoS and Deadline Awareness

MPXCP's explicit congestion control philosophy provides a flexible framework for designing a variety of capacity allocation strategies. The high-level approach is to assign different flows with different weights. Meanwhile, because we make every effort to keep routers stateless, MPXCP is a lightweight and scalable capacity allocation solution. We use two examples to show MPXCP's flexibility.

3.7.1 Quality of Service

Allocating bottleneck capacity proportionally to the price paid by each host pair is a desirable property [Popa et al. (2012)]. To achieve this, we change only the first control logic in Aggregated-Additive-Increase (Chapter 3.3.5) as follows: If $\phi > 0$, a switch allocates feedback to each packet such that the throughput increases for each flow is proportional to its price. This logic can be easily implemented by replacing the $cwnd_f$ field in the congestion header with $\frac{cwnd_f}{price_i}$, where $price_i$ is the price paid by s_i.

We use simulation to show that the above simple modification can allocate bottleneck

capacity to each flow proportionally to its price. Three flows are traversing the fence topology with prices of 1, 2, and 3. All of the flows start at $0s$. Between $0s$ and $5s$, each flow's throughput is proportional to its price. After 3-price flow stops at $5s$, the rest flows efficiently utilize the bottleneck and 2-price flow takes $\frac{2}{3}$ of the capacity. When 2-price flow stops at $10s$, the remaining 1-price flow grabs all the bottleneck capacity. Another observation is that MPXCP is highly responsive to flow dynamics and converges in a few RTTs every time the traffic pattern changes.

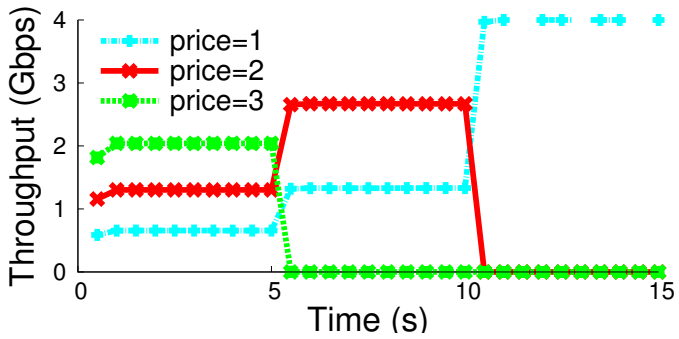


FIGURE 3.16: This figure shows MPXCP can provide different services by ensuring users' throughputs proportional to their prices. We inject three flows in the fence topology. The 3-price flow lasts from $0s$ to $5s$. 2-price flow lasts from $0s$ to $10s$. 3-price flow lasts from $0s$ to $5s$. The more expensive the price is, the larger throughput the flow gets.

3.7.2 Deadline Awareness

MPXCP is not designed to meet flow deadlines. However, MPXCP's explicit congestion control provides us with the flexibility to assign higher prices to the flows with tight deadlines. By doing so, MPXCP can indirectly prioritize flows and thus catch more deadlines. Besides, compared with other deadline-aware protocols [Hong et al. (2012)], MPXCP does not maintain any flow state along the path and thus significantly simplifies the overhead of switch upgrading. We use one example to show MPXCP's potential to catch tight deadlines.

We inject 30 $500KB$ flows into the fence topology. Twenty of them have the deadline of $25ms$. These flows are referred as type A. The remaining ten flows have the deadline of

$50ms$, which are referred as type B. If we use the original MPXCP to fairly allocate the capacity, each flow has the throughput of $\frac{4Gbps}{30} = 0.13Gbps$ and finishes at time $\frac{500KB}{0.13Gbps} = 30ms$. Type A flows miss their deadlines. Instead, if we assign each flow with the price of $\frac{size}{deadline}$, *i.e.*, assigning A flows with the price of $\frac{500KB}{25ms}$ and assigning B flows with the price of $\frac{500KB}{50ms}$, then an A flow's throughput is twice as much as a B flow. After delivering all the A flows within $25ms$, the remaining B flows grab all the capacity and are delivered at $30ms$. We meet all deadlines.

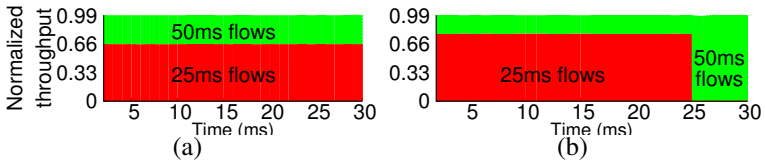


FIGURE 3.17: This figure shows that by assigning larger weights to deadline-constrained flows, MPXCP can reduce the fraction of missed deadlines. There are 30 $500KB$ flows. 20 of them have the deadline of $25ms$ and 10 of them have the deadline of $50ms$. As shown in Figure 3.17(a), if all the 30 flows fairly share the bottleneck, all the $25ms$-constrained flows miss their deadlines. However, as shown in Figure 3.17(b), if we assign each flow with the price of $\frac{size}{deadline}$, all the 30 flows meet their deadlines.

3.8 Related Work

XCP [Katabi et al. (2002)] considers efficiency and fairness as two independent aspects of congestion control. MPXCP follows the same philosophy. Furthermore, MPXCP needs to carefully allocate the aggregated feedbacks among multiple sub-flows. MPXCP chose to send more traffic along less congested paths and keep sufficient probing traffic along congested paths.

Another directly related work is TeXCP [Kandula et al. (2005b)]. It also extends XCP's control logic into multipath topology. However, because it focuses on intra-domain traffic engineering, the traffic demand is not elastic. This fundamentally different assumption makes TeXCP's congestion control law not applicable to multipath transport protocols.

3.9 Summary

We design and implement *Explicit Multipath Congestion Control Protocol* (MPXCP) for both datacenter networks and multi-homed wireless devices. MPXCP outperforms multipath TCP in four aspects: faster convergence, better efficiency despite different delay-bandwidth products, better fairness despite different RTTs, and negligible queue size. MPXCP is resilient to network dynamics. Its stateless implementation supports multiple bandwidth allocation policies.

4

NetPilot: Automating Failure Mitigation in Datacenter Networks

4.1 Introduction

The growing demand for always-on and rapid-response online services has driven datacenter networks (DCNs) to an enormous size, often comprising tens of thousands of servers, links, switches, and routers. To reduce capital expenses and increase overall system reliability, datacenter (DC) designers are increasingly building their networks using broad layers of many inexpensive commodity hardwares instead of large chassis switches. However, as the number of devices grows, the failure of network devices becomes the norm rather than the exception.

Diagnosing and repairing DCN failures in a timely manner has become one of the most challenging DC management tasks. Traditionally, network operators follow a three-step procedure to react to network failures: 1) detection; 2) diagnosis; and 3) repair. Diagnosis and repair are often time-consuming because the sources of failures vary widely, from faulty hardware components to software bugs to configuration errors. Operators must sift through many possibilities just to narrow down potential root causes. Even though auto-

mated tools exist to help localize a failure to a set of suspected components [Bahl et al. (2007); Kompella, R.R and Yates, Jennifer, and Greenberg, Albert and Snoeren, Alex (2005)], operators still have to manually diagnose the root cause and repair the failure. These diagnosis and repair sometimes require third-party device vendors' assistance, further lengthening the failure recovery time. Because of the above challenges, it can take a long time to recover from disruptive failures even in well-managed networks. For instance, in April 2011, a failure in Amazon's AWS service impaired the operations of many cloud services for hours [Amazon (2011)].

Realizing the problem above, we take a fundamentally different approach to tackle the failure recovery problem in large-scale DCNs. Specifically, we advocate a four-step process to react to failures: 1) detection; 2) *mitigation*; 3) diagnosis; and 4) repair. We argue that it is more important to mitigate failures than to fix them in real-time. Here "mitigate" means taking action(s) that alleviate the symptoms of a failure, possibly at the cost of temporarily reducing spare bandwidth or redundancy. Timely and effective failure mitigation enables a DCN to operate continuously even in the presence of failures, and allows operators to dive directly into failure diagnosis and repair.

This chapter presents **NetPilot**, an *automated system* that adopts our four-step process to *quickly* mitigate failures in a large-scale DCN before operators diagnose and repair the root cause. NetPilot can significantly shorten the failure disruption time by mitigating failures without human intervention. It can also improve online user experience and lower potential revenue losses that stem from service downtime. Moreover, it can lower a DC's operational costs, as it reduces the number of emergent failures and the number of midnight calls to on-call operators.

A key observation that motivates NetPilot's design is that simple actions such as deactivation or restart, coupled with the redundancy that exists in a DCN (Chapter 4.2), can effectively mitigate most types of failures in a DCN. DCNs often have extra links and switches to accommodate peak traffic load and device failures. In many cases, simple ac-

tions such as deactivating or restarting an offending component can mitigate failures with little impact on the network's normal functions. This observation differentiates NetPilot's design from conventional failure diagnosis and repair approaches, which require detailed failure-specific knowledge.

Since NetPilot has mitigation as its end goal, it can operate without human intervention and without knowing the precise failure root cause. NetPilot automatically mitigates failures in DCNs through a trial-and-error approach. First, it detects a failure and identifies the set of suspected faulty components. Then, it intelligently iterates through the suspected devices and applies mitigation actions on them one by one, until it mitigates the failure or has exhausted all possible actions.

Realizing NetPilot requires overcoming two key technical challenges. First, when mitigation actions are applied blindly, they can actually further compromise the health of the network, *e.g.*, deactivating a switch may overload other switches during peak hours. NetPilot avoids this problem by employing an Impact Estimator to accurately predict the impact of mitigation actions and executing the actions within a pre-defined safety margin.

Second, although NetPilot can safely try numerous mitigation actions before successfully mitigating a failure, an excessive number of trials will unnecessarily lengthen the failure mitigation process. NetPilot addresses this challenge with an optimized mitigation planner that uses failure-specific information to localize a failure to the most likely faulty components, and orders the sequence of mitigation actions according to their potential benefits.

To the best of our knowledge, NetPilot is the first automated failure mitigation system for DCNs. Our contributions are:

- We study and classify the high-impact failures in production DCNs over a six-month period, and find that we can mitigate most of those failures by simple actions such as restart or deactivation. We also find that there is sufficient redundancy in a DCN

to accommodate the impact of mitigation actions (Chapter 4.2, 4.3).

- We design and implement NetPilot (Chapter 4.4, 4.5) and deploy it in a testbed that resembles a real DCN topology. We also conduct simulations using data from a production DCN to evaluate NetPilot at a large scale. We experimentally validate the accuracy of the Impact Estimator, and find that it offers an error rate of less than 8%

- We use NetPilot to automatically mitigate three types of high-impact failures that operators often encounter in production DCNs. Besides reducing operational overhead, NetPilot decreases the median mitigation time from 2 hours to 20 minutes compared to current operational practice, significantly shortening a failure's impact on online services.

- We justify the design choices made in NetPilot. Compared to simple heuristics, NetPilot can succeed with fewer trials while maintaining safe operating conditions in the network.

4.2 Redundancy in Datacenter Networks

NetPilot's design is motivated by the observation that modern DCNs have a significant amount of redundancies at the device level, protocol level, and application level. NetPilot takes advantage of these redundancies to automatically mitigate failures.

A DCN design must support tens of thousands of servers with high bandwidth and at low cost. Solutions have already converged to one design paradigm: using many inexpensive commodity devices to scale up capacity and to reduce cost while deploying various types of redundancy to combat unreliability [Hoelzle and Barroso (2009)]. We describe three redundancy types below.

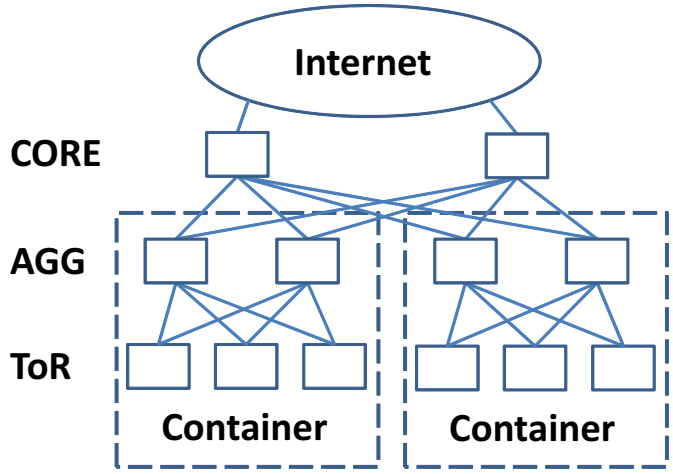


FIGURE 4.1: An example scale-out DCN topology.

4.2.1 Device-Level Redundancy

A typical modern DCN design uses a scale-out topology to create multiple redundant paths between host pairs. Scale-out topologies such as a Fat-Tree [Al-Fares et al. (2008)] and Clos [Dally and Towles (2003)] can achieve full bi-section bandwidth using commodity switches that often have low port density.

Figure 4.1 shows an example scale-out topology. This topology has three layers: top-of-rack (*ToR*), aggregation (*AGG*), and core (*CORE*). A *container* is a management unit and also a replicable building block sharing the same power and management infrastructure. A *CORE* switch connects to multiple containers. For ease of exposition, we use *ToR*, *AGG*, and *CORE* to refer to a switch at the ToR, aggregation, or core layer, respectively.

This scale-out topology provides multiple paths between any two servers. Such path diversity makes the network resilient to single link, switch failure. For example, in Figure 4.1, deactivating a single link or device, with the exception of a *ToR*, will not partition the network. Even a failed *ToR* will isolate only a small number of servers connected to it.

4.2.2 Protocol-Level Redundancy

DCNs also use various protocols to meet traffic demands under failure conditions. There are three practical technologies providing load balancing and fast failover at the link, switch, and path levels. These technologies are widely available in commodity switches.

Link Aggregation Control Protocol (LACP) abstracts multiple physical links into one logical link and transparently provides high aggregate bandwidth and fast failover at the link level. We use Link Aggregation Group (LAG) to refer to the above logical link. LACP provides load balancing by multiplexing packets to physical links by hashing packet headers. Some LACP implementations allow a LAG to start from one switch but to end at multiple switches.

Virtual switch is a logical switch composed of multiple physical switches. A network can use a virtual switch at the link or the network layer to mask the failures of physical switches.

A virtual switch tolerates faults at the network layer through an active/standby configuration. One switch is designated as the primary while the standby switch remains silent until it detects that the primary has failed. Two common implementations of network layer virtual switches are the virtual redundancy router protocol (VRRP) [S. Nadas, Ericsson (2010)] and hot standby router protocol (HSRP) [T. Li, B. Cole, P. Morton, D. Li (1998)]. Both VRRP and HSRP can be configured to provide load balancing.

A virtual switch at the link layer allows the physical switches to simultaneously forward traffic. Generically called Multi-Chassis LAG (MC-LAG), Virtual Port Channel (VPC) [vpc (2012)] and Split Multi-link Trunking [IEEE (2000)] are two common implementations.

Full-mesh *CORE*s refer to the full-mesh interconnections between *CORE*s and containers, *i.e.*, every container connects to every core switch [Niranjan Mysore et al. (2009); Greenberg et al. (2009)]. The ECMP [Hopps (2000)] routing protocols in full-mesh-

*CORE*s topologies provide load balancing and fast failover for traffic between containers.

4.2.3 Application-Level Redundancy

Modern DCNs also deploy application-level redundancy for fault tolerance. A common technique to increase failure resilience at the application level is to distribute applications under multiple *ToR*s. Therefore, deactivating any switch including a *ToR* is unlikely to have more than an ephemeral impact on the applications.

Table 4.1: This table categorizes the high-impact failures in several production DCNs over a six-month period. All failures listed here are either visible to users or impact revenue [Wu et al. (2012)].

Category	Detection	Mitigation	Repair	Percentage
software 21%	link layer loop	deactivate port	update software	19%
	imbalance triggered overload	restart switch		2%
hardware 18%	FCS error	deactivate port	replace cable	13%
	unstable power	deactivate switch	repair power	5%
unknown 23%	switch stops forwarding	restart switch	n/a	9%
	imbalance triggered overload	restart switch		7%
	lost configuration	restart switch		5%
	high CPU utilization	restart switch		2%
configuration 38%	errors on multiple switches	n/a	update configuration	32%
	errors on one switch	deactivate switch	update configuration	6%

4.3 Redundancy Warrants Automated Failure Mitigation

In this section, we analyze and classify the failure records in a six-month period from several production DCNs (DCNs$_p$). We then show that most failures are easy to detect, but difficult to diagnose or repair. However, we can mitigate them using simple actions such as deactivation or restart.

We obtained six months of failure records for DCNs$_p$, all of which were manually created by operators and contain detailed descriptions of critical failures. Network operators consider a failure critical if it is either visible to users or impacts revenue. Therefore each record represents a failure that required immediate investigation and response. The fields

of interest from these records are: the data sources used to detect the failure, the techniques used to mitigate the failure, the final actions taken to repair the failure, and the start and end times.

Table 4.1 classifies DCNs$_p$'s critical failures. We find that the single largest source of failures are misconfigurations. Prior research made similar findings in other contexts, such as ISP networks [Shaikh et al. (2002)]. Misconfigurations are common in DCNs due to the inherent complexity in managing configuration files in large-scale networks. Many of these misconfigurations, *e.g.*, incorrect ACL (Access Control List) rules, lead to lack of connectivity between certain hosts. Host-to-host pings can detect these misconfigurations, but fixing them is challenging and usually requires operators to manually debug the problems.

The next common type of failure is device software failures. One such example is a malfunctioning hash function that results in uneven link utilization among the physical links in a LAG. In some situations, the uneven utilization is so severe that one of the physical links becomes overloaded and discards packets. We can detect this type of failure by comparing the utilization of each link in the LAG. While we can detect software failures, diagnosing their root causes is generally nontrivial since operators know little about the inner workings of device binary code. Even for device vendors who have the source code, debugging the software is still challenging because failures cannot be easily reproduced in lab environments.

The third category of failures is hardware failures. Such failures occur frequently due to the large number of devices used in DCNs$_p$. For example, 13% of the failures are caused by a single problem, frame checksum (FCS) errors, which often significantly elevates host-to-host latencies. Like other failures, FCS errors can be detected by checking switch SNMP counters, *i.e.*, checking the number of corrupted frames received on each port. However, pinpointing the sources of FCS errors takes time because corrupted frames propagate throughout the network due to cut-through switching (§4.4.3). Even after iden-

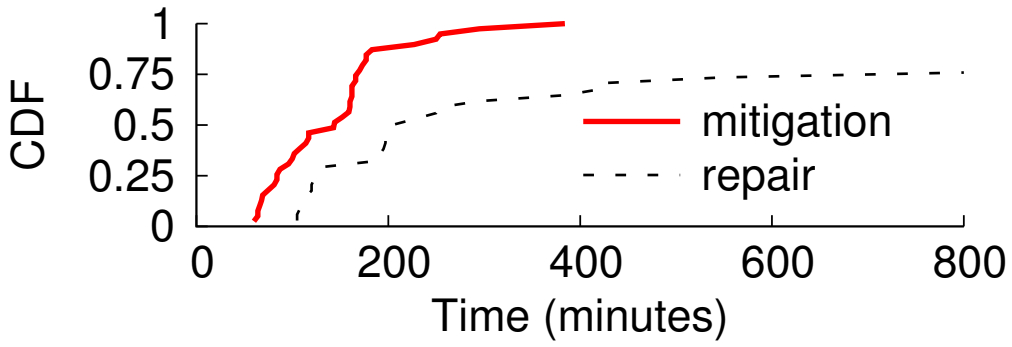


FIGURE 4.2: This figure shows the CDFs of how long it takes for DCNs$_p$'s operators to mitigate and to repair critical failures.

tifying the correct source, operators have to manually replace the corrupted cable.

The last category is failures whose sources are unknown. In some failures, switches have high CPU utilization while their traffic load is low. In other cases, switches may suddenly cease to forward traffic. These failures often cause abnormally high latency or packet losses. Due to the intermittent and unpredictable nature of these failures, they are difficult to reproduce, diagnose, and repair.

4.3.1 Time-Consuming Failure Recovery

From studying the types of failures in DCNs$_p$, we see that completely repairing a failure may require debugging of code, software update, or hardware replacement. Further, this process may also involve multiple parties: network operators, software developers, hardware engineers, and external vendors. It is therefore difficult to automate. Figure 4.2 shows the CDFs of failure mitigation and repair times in DCNs$_p$. As can be seen, the failure repair times can exceed several days or even weeks.

4.3.2 Simple Mitigation Actions are Effective

Table 4.1 shows that simple actions such as deactivating or restarting a switch or port can *mitigate* most types of failures in DCNs$_p$ before the root cause can be repaired. For

example, in the case of FCS errors, operators reduce latency by deactivating the corrupted links. In the case of overload triggered by load imbalance, operators restore load balance by restarting the offending switches.

In fact, operators of DCNs$_p$ already take *manual* mitigation actions to restore a network to a functioning state while diagnosing and repairing the failure. NetPilot's design goal of automating failure mitigation is partly motivated by the difficulty of manual mitigation. We compare the manual failure mitigation time with the repair time in Figure 4.2. As can be seen, the time it takes to mitigate failures (even manually) is much shorter than that to repair them. The median failure mitigation time is about two hours.

We note that not all failures can be mitigated by simple actions. Certain failures, *e.g.*, a global configuration error such as a misconfigured ACL, would require a network-wide reconfiguration to fix, and thus cannot be mitigated by restarting or deactivating a few offending devices. How to automatically mitigate those failures is beyond the scope of this paper.

4.3.3 Spare Capacity for Mitigation Actions

From the analysis above, we find that simple actions are highly effective in mitigating failures and also lead themselves to automation. An automated failure mitigation system can significantly reduce failure mitigation time, as well as the burden on operators.

However, one might be concerned that these simple mitigation actions may overload the network. To find out whether a DCN would have sufficient capacity for failure mitigation, we use the maximum link utilization after deactivating a component to measure the amount of spare network capacity. We use the Impact Estimator to carry out this computation. (We will describe Impact Estimator in detail in Chapter 4.4.2.)

We perform this computation using the traffic matrices aggregated over 10-minute intervals in one month for DCN$_p$, a single DCN in DCNs$_p$. Figure 4.3 shows the fraction of time intervals during which deactivating a link, a LAG, or a switch will not cause the

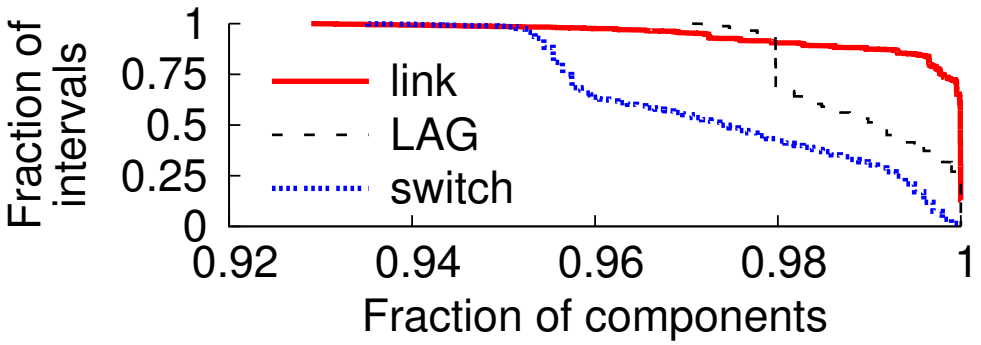


FIGURE 4.3: This figure shows the fraction of time intervals during which deactivating a link, a LAG, or a switch will not cause the maximum link utilization to exceed a target threshold (90%) versus the fraction of components for which this result holds.

maximum link utilization to exceed a target threshold (which is 90%) set by DCNs$_p$'s operators and the fraction of components for which this holds. As can be seen, 99% of the links can be deactivated in over 90% of the intervals without overshooting the target threshold. These numbers are slightly lower for LAGs and switches, because a LAG or a switch contains multiple links. Overall, there is sufficient redundancy in DCN$_p$ to tolerate a component deactivation most of the time.

4.4 NetPilot Design

NetPilot quickly mitigates failures without knowing their actual root causes in four steps. The first step (S_1) is *failure detection*, in which it constantly monitors the network and detects any potential failure. The second step (S_2) is *mitigation planning*. When NetPilot detects a failure, it will propose a set of suspected components, determine the appropriate mitigation actions, and order these actions based on the likelihood of success or potential impact. The third step (S_3) is *impact estimation*. To avoid taking any action that would further degrade network health, NetPilot estimates the impact of each action and discards the actions that are considered unsafe. The last step (S_4) is *plan execution*. NetPilot will successively execute each mitigation action. If an action successfully mitigates the failure,

NetPilot marks the failure as mitigated. Otherwise, NetPilot will roll back the action and try the next action.

In this section, we focus on two main technical challenges: impact estimation (S_3) and mitigation planning (S_2). We postpone the discussion of failure detection (S_1) and plan execution (S_4) to the next section. Note that impact estimation *must* be accurate in order for NetPilot to avoid actions that could further degrade network health. However, NetPilot can work properly even *without* precisely localizing a failure or ordering the mitigation actions.

4.4.1 Impact Metrics

A chief design goal for NetPilot is to avoid taking any mitigation action that could further degrade a DCN's health. Typically, for a given traffic matrix over a time interval T, we can assess a DCN's health via three metrics: *availability*, *packet losses* and *end-to-end latency*. The availability and packet losses of a DCN can be quantified by the fraction of servers with network connectivity to the Internet ($online\_server\_ratio$) and the total number of lost packets ($total\_lost\_pkt$) during the interval T respectively. Quantifying latency is tricky because it is difficult to predict how intra-DC network latency would change after a mitigation action. Given this problem, we use the maximum link utilization (max_link_util) across all links during the interval T as an indirect measure of network latency. Because the propagation delay is small in a DCN (no more than a few milliseconds), low link utilization implies small queuing delay and thus low network latency. Next, we will explain how to predict these metrics after a mitigation action.

4.4.2 Estimating Impact

The Impact Estimator aims to estimate a mitigation action's impact on a DCN. Answering this question is crucial for ensuring the safety of mitigation actions. The Impact Estimator takes an action A and a traffic matrix TM as two input variables and computes the expected

impact of A under TM. Since it is straightforward to compute $online\_server\_ratio$ given a network topology, we focus on estimating max_link_util and $total\_lost\_pkt$ in the rest of the discussion.

We can get the max_link_util and $total\_lost\_pkt$ by collecting SNMP counters in a DCN. However, predicting these two metrics after a mitigation action is nontrivial because the action could change the traffic distribution in the network. In a DCN with no centralized routing control, we cannot precisely predict how packets are routed to their destinations, unless we know all the packet headers, forwarding tables, and load balancing hash functions.

Our approach to address this challenge is based on two important facts that shape the traffic distribution in DCNs. First, there are far more flows than the diversity of paths in DCNs [Greenberg et al. (2009); Kandula et al. (2009b)]. Second, hash-based flow-level load balancing is widely used at the link level, switch level, and path level in production DCNs [Hopps (2000); Greenberg et al. (2009)].

These two facts make packet headers and load balancing hash functions, which are difficult to obtain in real-time, unnecessary in predicting traffic distribution. Our intuition is that hashing many flows onto a relatively small number of paths leads to even load balancing [Greenberg et al. (2009)]. As a result, a coarse-grained TM plus forwarding tables should enable us to estimate the real traffic distribution with reasonably high accuracy.

We choose to represent a TM at the granularity of *ToR*-to-*ToR* traffic demands instead of server-to-server, because this representation dramatically reduces the size of TM while not affecting the computation of traffic distribution at the *AGG* or *CORE* layers.

Besides TMs, we also need the forwarding tables to know the next hops to any given destination. As explained in Chapter 4.2, a DCN typically follows a hierarchical structure with traffic traversing valley-free paths. This inspires us to infer the forwarding tables in a similar manner, as illustrated in Figure 4.4. In the first bottom-up iteration, every switch learns the routes to its descendant *ToR*s from its direct children. In the second top-

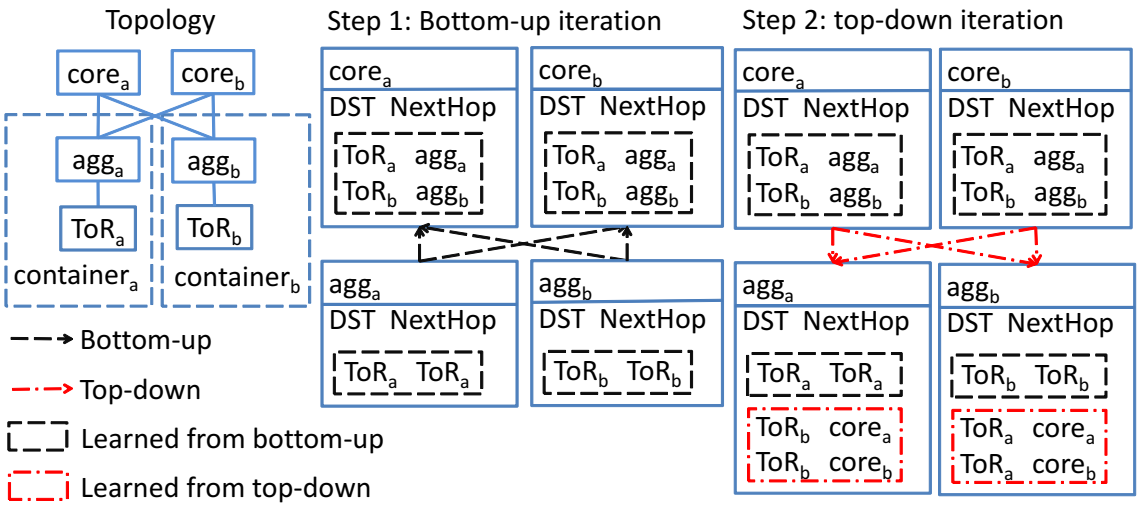


FIGURE 4.4: A switch learns the equal cost next hops to any *ToR* after two iterations in a hierarchical DCN topology.

down iteration, every switch learns the routes to the non-descendant *ToR*s. After these two iterations, every switch builds up the full forwarding table to any *ToR*s in the network.

Algorithm 5: $node.Forward(load)$

```
1: if load.dst == node
2:     return; // reach the destination;
3: nxtHops = node.Lookup(load.dst)
4: for each node n in nxtHops
5:     for each link l between node and n
6:         subload.dst = load.dst;
7:         subload.volume = load.volume/|nxtHops| × 1/|links between node and n|;
8:         n.Forward(subload);
```

We use the term $load$ to refer to the traffic demand between two *ToR*s. Algorithm 5 $node.Forward$ presents how a $node$ forwards a $load$ in detail. Line 3 returns all the next hops (*nxthops*) to a destination. Assuming even load balancing for traffic crossing adjacent levels in the network hierarchy, Lines 4-8 first evenly split $load$ among the *nxthops*, and then for each next hop, the traffic is evenly split among the physical links. The second traffic split is necessary due to the presence of LAGs (described in Chapter 4.2). By running this algorithm on each $load$ in TM and aggregating the contribution of each $load$

on each link, we predict all the link utilizations.

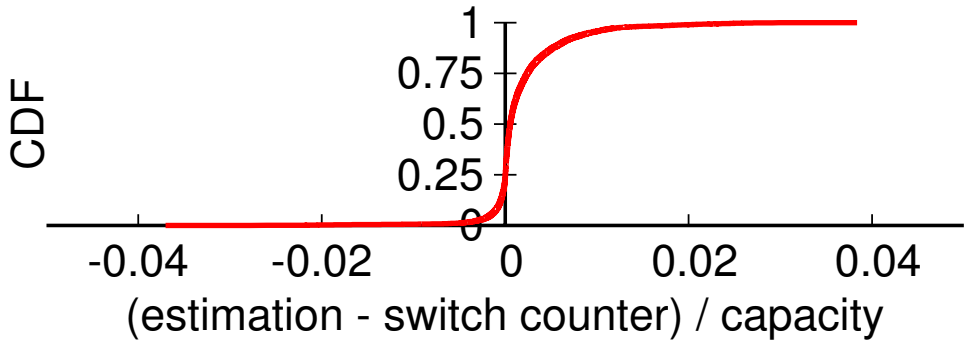


FIGURE 4.5: The differences between the link utilizations read from SNMP counters and those computed by Impact Estimator are within 4%.

We evaluate Algorithm 5 $node.Forward$ in an 8000-server production DCN to see whether the Impact Estimator is accurate. We log one-month's socket events on all the servers and aggregate the logs into *ToR*-to-*ToR* traffic matrices at a 10-minute granularity. We also collect the link utilizations during the same month via SNMP at a 10-minute granularity. Figure 4.5 shows the CDF of the relative differences between the estimated link utilizations and the measured ground truth. As can be seen, the Impact Estimator works very well, and has a maximum error rate of $\pm 4\%$.

So far, we have explained how the Impact Estimator works under a known network topology and TM. To predict the impact of an action, we need to know the new topology and TM after the action is committed. Although inferring the new topology is straightforward, predicting the new TM can be tricky because a mitigation action might affect the traffic demand from minutes up to days. For a restart action which takes only several minutes, we use the TM in the most recent time interval (*e.g.*, 10 minutes) to predict the action's impact during the restart period, assuming the TM is unlikely to change dramatically in such a short time. For a deactivation action that may last days, due to a faulty component needing to be replaced, we ideally desire to use the TM's in the future days to predict the impact during the deactivation period. However, traffic prediction is a research

topic by itself, and is beyond the scope of this paper. Instead, we use the TMs in the most recent n days before a deactivation action to predict the impact in the future n days, assuming that the traffic demands are stable over $2n$ days when n is small. In our evaluation (Chapter 4.6.5), we find that this simple heuristic works reasonably well.

4.4.3 Planning Mitigation

Given that NetPilot takes a trial-and-error approach toward failure mitigation, it needs a mitigation planner to localize suspected components and prioritize mitigation actions to minimize the number of trials. One simple solution is to use existing work [Kandula et al. (2005a); Bahl et al. (2007); Kandula et al. (2009a)] to localize failures and then iteratively try deactivating or restarting the suspected components. Although this simple, failure-agnostic solution might work, we choose to develop a mitigation planner that uses failure-specific knowledge to achieve finer-grained localization and more meaningful ordering of mitigation actions (*i.e.*, based on success likelihood). This in turn leads to fewer trials and shorter mitigation times. The downside is that NetPilot needs a planning module for each type of failure. However, we consider this trade-off worthwhile since there are relatively few types of critical failures in DCNs (as shown in Table 4.1).

In the presentation that follows, we first describe in detail mitigation planning for three types of failures: *FCS errors*, *link-down*, and *uneven-split*. We will then discuss the other failure types (listed in Table 4.1), which are easier to handle compared to these three types.

Frame Checksum Errors

Many of the links in a DCN are optical. When foreign material such as dust gets between an optical cable and its connector, the packets traversing the optical link can suffer bit flips. This causes a frame to mismatch its checksum. As shown in Table 4.1, FCS errors occur frequently in DCNs$_p$ and can significantly degrade performance. Figure 4.6 shows how a corrupted link impacts the application-level latency in DCN$_p$. Around 12:00pm, a

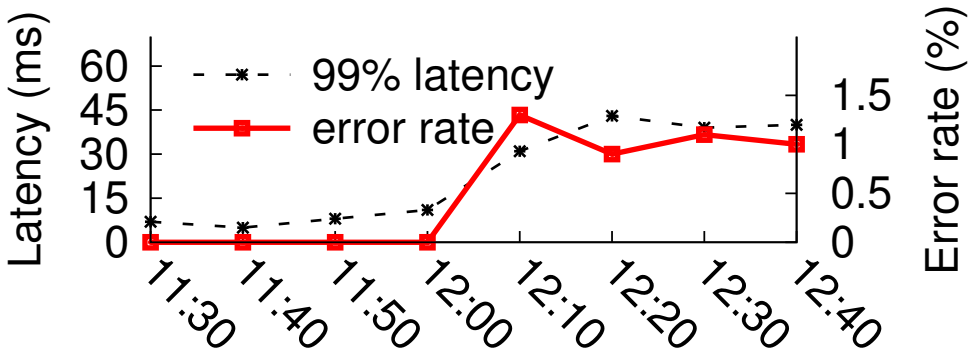


FIGURE 4.6: The 99-percentile application-level latency increases significantly due to one corrupted link in DCN_p.

link connecting a *ToR* and an *AGG* switch began to corrupt 1% of all packets. This 1% corruption rate leads to a 4.5 times increase in the 99^{th} percentile latency for applications running under that *ToR*.

Although replacing the faulty cable is the ultimate solution, this could take days depending on staff availability. Operators can mitigate the failure by disabling the faulty link before it is replaced. However, identifying the faulty link is challenging due to the wide use of cut-through switching [cut (2008)] in DCNs. Because cut-through switches start forwarding a frame before they can verify its checksum, switches can distribute corrupted packets across the network before the corrupted packets are detected locally.

Figure 4.7 exemplifies how cut-through switching affects FCS errors in DCN_p. The operators began to observe many corrupted packets around hour 6.5, when 28 ports' error rates exceed 1%. Over the next 3.5 hours, operators were busy deactivating the suspected ports one-by-one to determine the faulty links. Finally around hour 10, they found and deactivated the two offending ports and the link error rates returned to normal afterward.

Mitigating FCS errors. Our solution is based on two observations. First, *errors are conserved* on cut-through switches that have no faulty links, *i.e.*, the number of incoming corrupted packets should match the number of outgoing corrupted packets. This observation holds because packet losses are uncommon and broadcast/multicast packets account

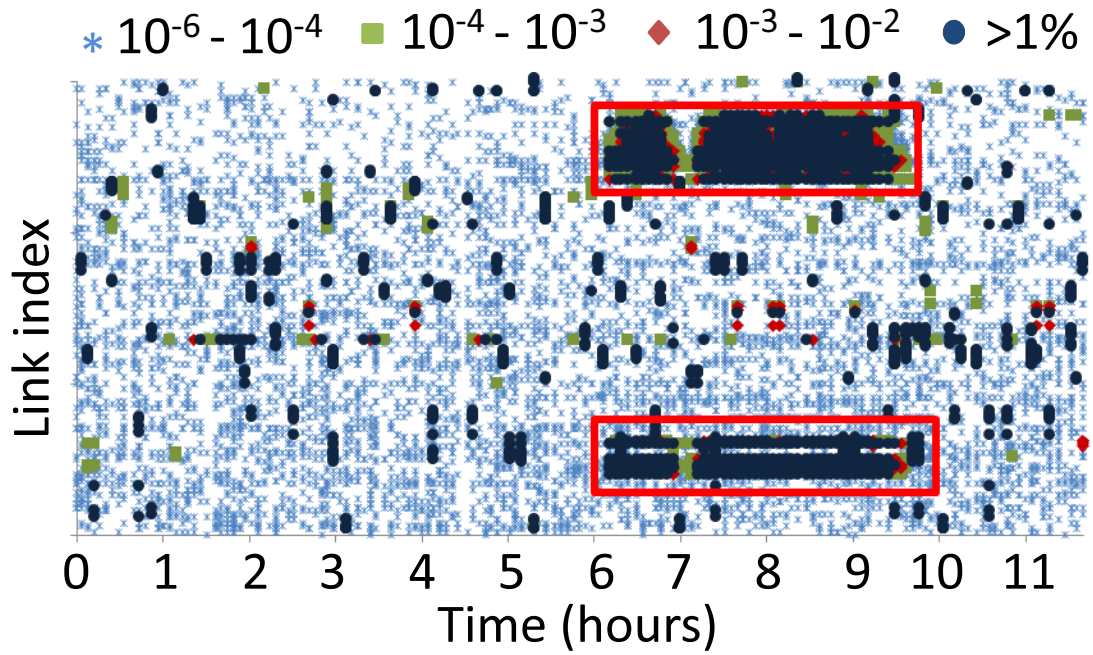


FIGURE 4.7: Each point shows a link's error rate. Darker dots indicate higher error rates. We highlight two areas that have the highest error rates (>1%) in 28 links. They are caused by two corrupted links in DCN_p, and it took the operators 3.5 hours and 11 trials to deactivate the two offending links.

for only a tiny fraction of the total traffic in DCNs_p. Second, the error rate of each faulty link is small and the number of simultaneous faulty links is small. Therefore, it is unlikely that multiple faulty links contribute to the corruption of one packet.

Based on these two observations, we design an FCS error propagation model to localize faulty links. We use x_l to denote link l's corruption rate, p_l and e_l for the total number of packets and the number of corrupted packets traversing l respectively, and m_{kl} for the fraction of packets coming from link k that also traverse link l. Note that the number of corrupted packets coming from link l is equal to the number of packets corrupted by l plus the number of packets corrupted by other links that traverse l. By ignoring the packets

corrupted by multiple links, we have:

$$e_l = \sum_{k \neq l} p_k \; x_k \; m_{kl} + p_l \; x_l \tag{4.1}$$

We use the same technique as that of the Impact Estimator to compute m_{kl}. e_l, p_k and p_l can be obtained from SNMP counters. Thus, the linear equations (4.1) provide the same number of constraints as the number of variables (x_l's). If we get a unique solution, the faulty links are those with non-zero x_ls. If the solutions are not unique, we simply pick one with the smallest number of non-zero x_ls based on the fact that the number of simultaneous faulty links is usually small. Our evaluation shows that this approach works well in practice with very few false positives (§4.6.3).

Link-down and Uneven-split

Even when the network has the capacity to handle the offered load, link overloading may still occur due to load imbalance or link failure, leading to packet losses and high latencies in DCNs.

Link-down: When one link in a LAG_x is down, the LAG_x will redistribute the traffic to the remaining links. Since this process is transparent to higher layer protocols, traffic demands remain the same over LAG_x. Thus, LAG_x can become overloaded. One mitigation strategy is to deactivate the entire LAG_x and have the traffic re-routed via other LAGs to the *nxthops* (defined in §4.4.2). Another strategy is to deactivate all the LAGs (including LAG_x) to the *nxthops* and re-route the traffic to other switches.

Uneven-split: Due to software or hardware bugs, a switch may unevenly split traffic among the *nxthops* or the links in a LAG. In DCNs$_p$, we sometimes observe extreme traffic imbalance such as when one link in a lag carries 5Gb/s more traffic than any of the other links in the LAG. While the exact root causes might be unknown, operators have found that restarting the LAG or switches on either end rebalances the traffic (at least for some period of time).

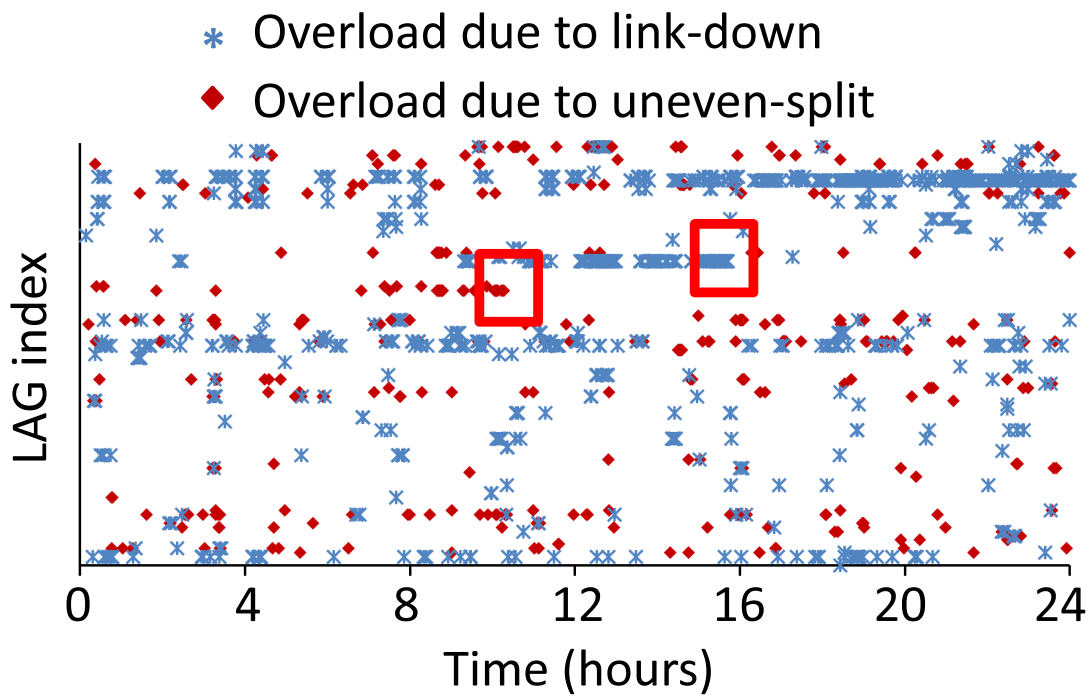


FIGURE 4.8: Each point indicates a link overloading incident in a LAG caused by load imbalance in a production DCN. We highlight two incidents. The first, which occurred around hour 10, was due to uneven-split and was mitigated by restarting a switch. The other, which occurred around hour 16, was due to link-down and was mitigated by deactivating a LAG.

Figure 4.8 illustrates that both types of failures are common in DCN_p. We collect all the link utilizations every 10 minutes for one day in DCN_p. In this figure, each dot represents a LAG that meets the following two conditions in a 10-minute interval: 1) at least one link is overloaded (utilization $> 90\%$); and 2) at least one link is broken (*link-down*) or the difference between the maximum and mean link utilizations exceeds 5% (*uneven-split*). We choose 5% as the load imbalance threshold because Figure 4.5 not only suggests that accurate impact estimation is feasible but also suggests that large load variance ($> 5\%$) within a LAG is a strong indication of the traffic imbalance problem. We highlight two incidents in Figure 4.8: one *uneven-split* failure mitigated by a switch restart

around hour 10 and another *link-down* failure mitigated by a LAG deactivation around hour 16.

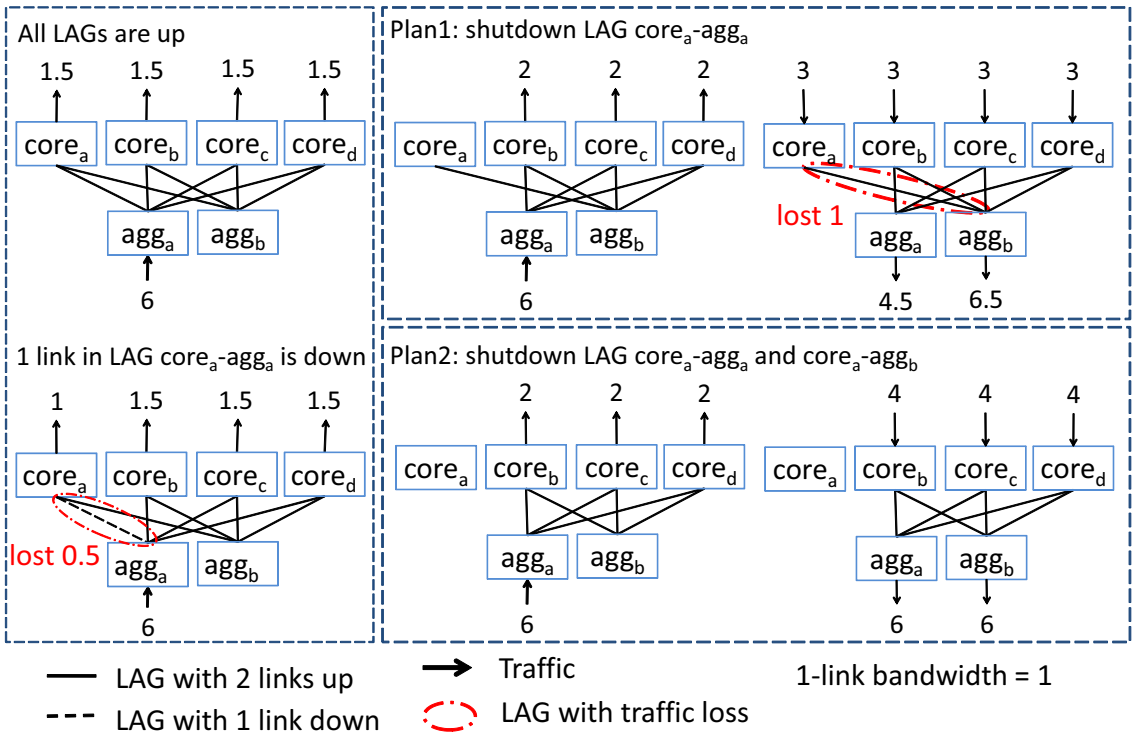


FIGURE 4.9: The first plan deactivates one LAG but still causes downward traffic loss. The second plan deactivates two LAGs without causing any traffic loss.

Mitigating a *link-down* or *uneven-split* requires some care due to the complexity of the traffic matrix and topology, as exemplified in Figure 4.9. Each pair of switches is connected by a LAG consisting of two physical links with a combined capacity of two units. There are six units of upward traffic from agg_a to the $cores$ and twelve units of downward traffic from $cores$ to $aggs$. Suppose one link between agg_a and $core_a$ is down, halving the corresponding LAG capacity, resulting in 0.5 unit of upward traffic loss. One obvious mitigation strategy (Plan 1) is to deactivate the entire LAG between agg_a and $core_a$. Although this prevents the upward traffic loss, it causes one unit of downward traffic loss between $core_a$ and agg_b. The correct strategy is to deactivate the LAG between

$core_a$ and agg_b as well (Plan 2). This will shift the downward traffic via $core_a$ to the other $cores$ and prevent traffic loss in both directions.

Mitigating *link-down*. NetPilot mitigates *link-down* failures by estimating the impact of all possible deactivation actions and carrying out the ones with the least impact, *i.e.*, minimizing maximum link utilization. Because a link could be down for n days, NetPilot needs to estimate an action's impact during the downtime. To do so, NetPilot uses the traffic matrices of the most recent n days (Chapter 4.4.2) as an approximation. Such a computation is difficult for human operators to perform because the number of mitigation actions and traffic matrices to consider in concert could be quite large.

Mitigating *uneven-split*. NetPilot mitigates *uneven-split* failures by restarting LAGs or switches. To limit the temporal interruptions during restarts, NetPilot prioritizes the restart sequence based on a restart's estimated impact, while also assuming a component cannot carry any traffic during restart. Since restarting one component usually takes only a few minutes, NetPilot uses the traffic matrix in the most recent time interval (*e.g.*, 10 minutes) as an approximation of the traffic matrix during the restart. After exhaustively calculating the impact for every possible restart, the planner will first carry out the action with the least estimated impact. If this action does not mitigate the failure, the planner will re-prioritize the remaining options based on the latest traffic matrix.

Other Types of Failures

FCS error, *link-down*, and *uneven-split* are by no means all the failures that NetPilot can mitigate. We carefully review all the critical failures in Table 4.1 and find 62% of them can be localized via available data sources (such as SNMP counters and syslogs) and can be mitigated via deactivation or restart. The only exceptions are the failures due to configuration errors (38%). Although configuration errors on a single switch can be mitigated by deactivating the misconfigured switch, identifying if a configuration error involves one

or multiple switches still requires human intervention. We briefly discuss how to apply NetPilot to mitigate other failures:

Link layer loop: Due to switch software bugs, link layer protocols sometimes never converge and cause severe broadcast storms. This failure can be localized by identifying the switches which become suddenly overloaded but experience little traffic demand increase. The mitigation strategy is to deactivate one of the afflicted ports or switches to restore a loop-free physical topology.

Unstable power: Failures due to unstable power are localized by searching syslogs for unexpected switch-down events. They can be mitigated by deactivating the switches impacted by unstable power.

Failures due to unknown reasons: Such failures account for 23% of all critical failures. Even if their root causes are unknown, they can be easily localized to a single switch and mitigated by a restart. For example, a switch that stops forwarding can be identified once the difference between its received and delivered bytes exceeds a threshold. It is also straightforward to identify a switch that loses its configuration or suffers from high CPU utilization.

4.5 NetPilot Implementation

NetPilot's primary implementation challenge is reliability. As a failure mitigation system, NetPilot itself must be robust to failures. We build NetPilot as a pipeline of five independent processes, as shown in Figure 4.10. These processes include a failure detector (Chapter 4.5.1), failure aggregator (Chapter 4.5.2), planner (Chapter 4.5.3), impact estimator (Chapter 4.5.4), and plan executor (Chapter 4.5.5). Each process records its relevant state to a replicated database so that the state can survive server crashes. Operators can also use the recorded state to determine ex post facto why NetPilot took specific actions.

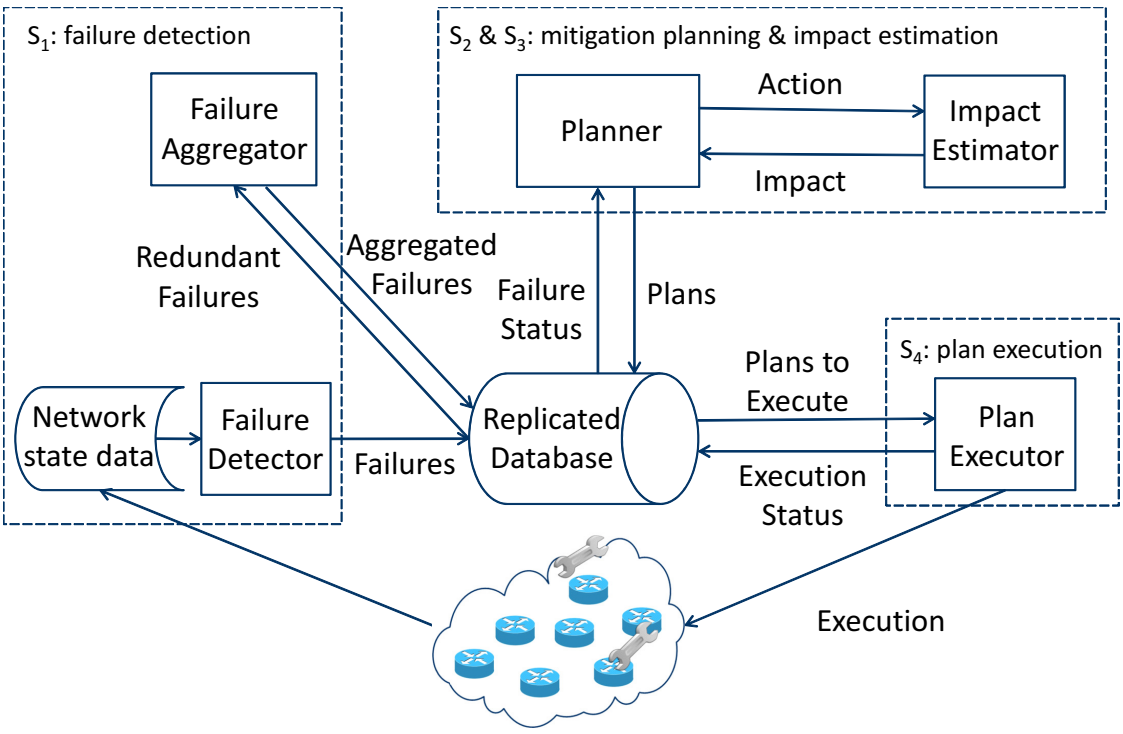


FIGURE 4.10: NetPilot implementation overview.

4.5.1 Failure Detector

The failure detector uses three data sources to detect failures: SNMP traps [snm (2002)], switch and port counters, and syslogs [Lonvick (2001)]. The detector then applies failure-specific criteria to evaluate whether a failure has occurred. For example, the failure detector looks at the *bytes-in* and *dropped-packets* counters of a port to determine if a link is overloaded. In our implementation, values from the above data sources are processed every five minutes.

When the failure detector detects a failure, it updates the database with the following information: the type of detected failure, data sources used to detect the failure, and the components that exhibit abnormal behaviors. Note that these components are not necessarily the faulty components, because the failure effects may propagate to healthy components, *e.g.*, a broken link may cause overload and hence packet losses at other links.

4.5.2 Failure Aggregator

Because the failure detector runs continuously and NetPilot takes a trial-and-error approach, we expect that the same failure will be detected multiple times before it is mitigated. NetPilot therefore needs a mechanism to decide whether a detected failure instance is a new or ongoing failure.

The failure aggregator compares a newly reported failure instance against all the ongoing failures recorded in the database. If it determines that the newly reported instance has not been mitigated before – determined by the failure type and components involved – it updates the database and marks the failure as ready for mitigation. If it has seen the failure and the planner is taking a mitigation action, it marks the instance as requiring no further action. If it has seen the failure and the planner has taken a mitigation action for the failure, it flags the failure as unsuccessfully mitigated. The planner may then try the next mitigation action if there is one available.

The failure aggregator does not remove the failure instance created by the failure detector, but simply marks that it has been processed so that an operator can examine the initial failure detection as well as the choices made by the failure aggregator later on.

4.5.3 Planner

The planner takes three steps to choose a mitigation action. First, it employs failure-specific modules to localize a failure to a set of suspected components. Second, it generates the appropriate mitigation actions against all suspected components. Third, it uses the impact estimator to estimate the impact of these actions, ranks them based on their impact or success likelihood, and then executes the best one. At the end of each step, the planner updates the database with its computation results for post-analysis.

4.5.4 Impact Estimator

The impact estimator implements the algorithm presented in Chapter 4.4.2. It uses the run-time DCN topology and historical TMs to compute $online\_server\_ratio$, max_link_util, and $total\_lost\_pkt$. We extract the run-time topology from device configurations and running state (*i.e.*, up/down). It includes both the physical and logical device connections such as a LAG that comprises multiple physical links and a virtual switch that comprises multiple physical switches. The traffic matrices are continuously collected via socket event logs on each server and are aggregated to ToR-to-ToR traffic matrices at a 10-minute granularity.

4.5.5 Plan Executor

Once the planner chooses a mitigation action, the *plan executor* is engaged to take the action. The executor translates the action into a series of commands recognized by switches. As the commands are vendor-specific, we create a vendor-specific *configlet* file that includes the commands for each mitigation action. A *configlet* file parameterizes configuration arguments such as port number, so it can be reused to take the same action on different switches or ports. We also implement a library that allows the executor to send commands to switches via both in-band and out-of-band channels. After an action is taken, the executor updates the database to record the time when the action was taken and whether the action was successfully applied to the switch.

4.5.6 Interactions with Operators

NetPilot is fully capable of mitigating failures without human intervention. Nonetheless, NetPilot is explicitly designed to record the inputs and outputs of each mitigation step in a manner that is readily accessible to operators. Operators can later examine the decisions at each step. This design helps them debug and understand counterintuitive mitigation actions. Moreover, it helps reveal failures that are repeatedly mitigated for only a short

period of time.

4.6 Evaluation

In this section, we show that NetPilot can quickly and automatically mitigate several types of critical failures in DCNs. After presenting our experimental methodology, we first conduct three end-to-end experiments to highlight that, compared with the approach used by today's operators in DCNs$_p$, NetPilot can mitigate failures faster and with less disruption. Then we conduct more detailed experiments to illustrate the three reasons why NetPilot outperforms the status quo: effective failure localization, accurate impact estimation, and action prioritization that minimizes disruption.

4.6.1 Experimental Methodology

We conduct four types of experiments to study various key aspects of NetPilot: 1) *failure-replay,* for failures with operation logs, we feed the device counters and traffic matrices when failures occurred in DCN$_p$ into NetPilot and compare NetPilot's actions with the actions that were actually taken by the operators; 2) *heuristic-replay,* for failures without operation logs, we feed the device counters and traffic matrices when failures occurred in DCN$_p$ into NetPilot and an *operator's approach* and compare their actions; 3) *testbed,* we inject traffic and failures into a testbed and compare the actions taken by NetPilot with those taken by operators; 4) *simulation,* we use large scale simulations to compare NetPilot and the operator's approach in the face of multiple simultaneous failures.

Operator's Approach

For failures with operation logs created by operators, we can directly compare NetPilot's actions with the actual operator's actions. For failures without operation logs, we build a model of how operators would mitigate the failures based on discussions with DCNs$_p$'s operators and our observations. We first carefully review the failures with operation logs

and enumerate the typical action sequences for the types of failures discussed in Chapter 4.4.3. We then have operators explain the reasoning behind each sequence of actions. We summarize the operator's heuristics for mitigating each type of failure and estimating impact below.

Estimating impact: When deactivating a link or switch, all of its traffic will fail over evenly among its redundant components. Components not in the same redundancy group have no change in traffic. Because manual computation is slow and error-prone, operators cannot afford to compute the traffic changes that are more than one hop away from the deactivated component.

Mitigating FCS errors: Identify the switches having significantly more corrupted packets of outgoing than incoming, and then deactivate their ports in the descending order of error rate.

Mitigating *uneven-split*: Try restarting LAGs first and then switches, because operators believe restarting a switch is likely to have greater impact.

Mitigating *link-down*: Compute the impact of each possible action under the latest traffic loads using the impact estimation heuristic above, and then execute the action with the least impact.

Experimental Setup

Our data for the failure-replay and heuristic-replay experiments come from DCNs$_p$, several large production DCNs using the scale-out topology for which operators log critical failures with details and mitigation actions. We collected six months of device counters via SNMP at 5-minute intervals. We also logged the socket events on all servers during the same six months and aggregated the logs into ToR-to-ToR traffic matrices at a 10-minute granularity.

As shown in Figure 4.11, our testbed is a mini version of DCN$_p$'s scale-out topology with all the important characteristics preserved. It has a hierarchical structure with two

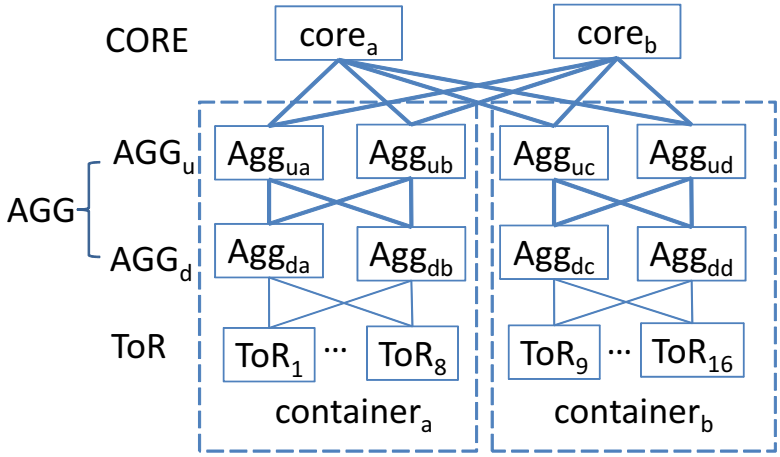


FIGURE 4.11: The testbed topology.

containers. The connections between containers and *CORE*s form a full mesh running BGP ECMP for load balancing. We use multi-chassis LAG to virtualize the AGG_d switch pairs and VRRP to virtualize the AGG_u switch pairs in the same container. Each connection above AGG_d is a LAG with four 10Gb/s physical links and each connection between a *ToR* and an AGG_d is one 10Gb/s physical link. Unlike the topology in Figure 4.1, the testbed topology has 1) two *AGG* levels: AGG_u and AGG_d, and 2) one traffic generator instead of multiple servers under each ToR. The traffic generators can either inject arbitrary traffic or replay real server traffic traces captured in DCN_p.

Our simulator for the simulation experiments use the same topology and traffic matrix data as the failure-replay experiments. However, we inject hypothetical failures and compare the actions taken by NetPilot and the operator's approach.

4.6.2 End-to-End Failure Mitigation

In this section, we compare the state-of-the-art manual failure mitigation by operators with NetPilot for three failure incidents in DCNs_p. We are limited by the number of available operation logs to conduct a large-scale comparison. Nevertheless, from the failure incidents we examined, we expect that NetPilot's improvement shown in these examples is

highly representative.

FCS error. Figure 4.7 shows the timeline of operators mitigating an FCS error incident. The operators iteratively tried deactivating links that had a significant error rate. Without an Impact Estimator, they performed manual calculations to ensure the remaining links would not become overloaded when they deactivated a link. It took a team of experienced operators nearly 3.5 hours and 10 unsuccessful trials to deactivate the two faulty links. In contrast, a failure-replay experiment shows that NetPilot can pinpoint and deactivate the two troublesome links in less than 15 minutes without any human intervention.

Overload due to *link-down*. Figure 4.8 depicts an incident of overloaded link caused by *link-down*. This incident persisted for almost 6 hours before the operators mitigated it by deactivating a LAG. The mitigation action was repeatedly delayed because the operator's manual impact estimation was inaccurate and informed them that deactivating the LAG would be worse than taking no action at all. After six hours of continually re-running their impact estimation, the operators deactivated the LAG around hour 16.

In contrast, a failure-replay experiment shows that NetPilot finds an alternative action that would have mitigated the incident soon after the failure was detected, which is almost 5.5 hours ahead of the operator's action. This is possible only because NetPilot's Impact Estimator is faster and more accurate and thus can exhaustively explore all options. This improved accuracy allows NetPilot to take actions that operators would have erroneously excluded.

Overload due to *uneven-split*. A known bug in the software that runs on the AGG_us causes them to occasionally stop generating routing updates. This in turn causes the *CORE*s to cease forwarding traffic to the afflicted AGG_us even though the links are up. Because it is difficult to consistently reproduce the exact same scenario in a testbed experiment, we emulate it by misconfiguring an AGG_u's ACL to block the TCP connections to the *CORE*s. Under this setting, *CORE*s' routing table entries with the afflicted AGG_u as the next hop will expire and most of the traffic will be sent to the other AGG_u in the same

afflicted container.

NetPilot detects this problem as an *uneven-split* incident and attempts to mitigate it by restarting switches. Because we only install the ACL rules in the running configuration and not the startup configuration, the problem will be mitigated upon switch restart.

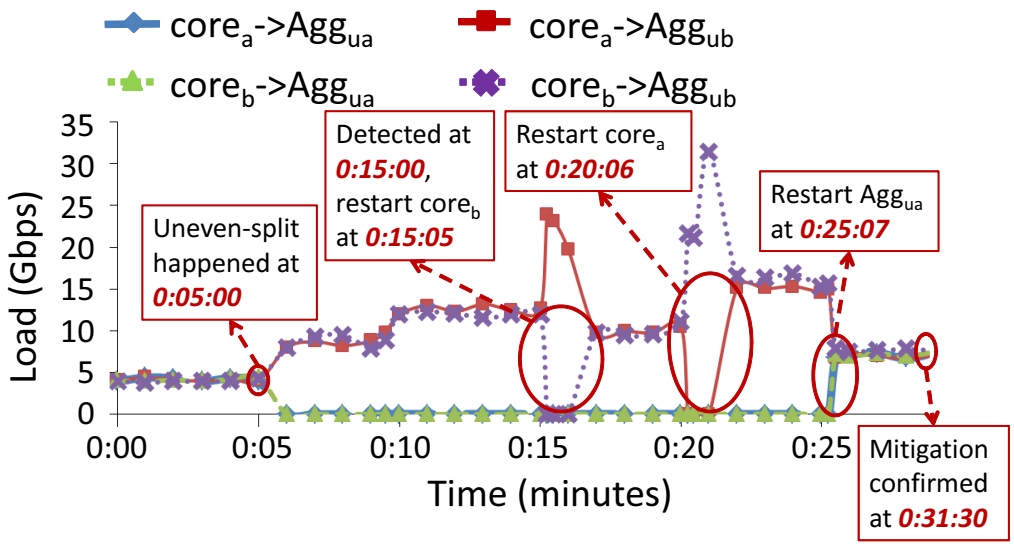


FIGURE 4.12: NetPilot mitigates link overloading due to *uneven-split* after restarting three switches.

Figure 4.12 shows the end-to-end mitigation process. We inject the *uneven-split* failure at minute 5 of the trace. Because switch counters are pushed to NetPilot once every 5 minutes, NetPilot responds at a granularity of 5 minutes. The first problem is identified at minute 15 because the failure aggregator waits for two consecutive counter values before declaring a failure. Approximately 5 seconds later, the planner finishes generating a list of possible actions, estimating the impact of each action via the Impact Estimator, and executing the one with the least impact. When the next set of counters gets pushed to NetPilot, the planner notices that the problem has not been mitigated and starts the next round of planning, excluding the actions that have already been taken. Five minutes later, the planner again notices the failure persists. As a result, it starts a third round of planning

and mitigates the failure after restarting the third switch. Even though the first two actions are incorrect, NetPilot can mitigate the failure in approximately 20 minutes, which is still much faster than engaging the operators and manually restarting the suspected switches.

4.6.3 Fine-grained Failure Localization

In this section, we show that NetPilot can expedite the mitigation process using its fine-grained failure localization. We compare NetPilot's FCS model described in §4.4.3 against three alternative approaches: 1) *Greedy deactivation*, in which links are deactivated in the order of decreasing error rate; 2) *Operator's approach*, which is described in §4.6.1; and 3) *Exhaustive search*, in which we try deactivating every possible combination of the links on the switches that violate the error conservation constraint, from one up to three links in each combination, in the order of decreasing combined error rate. If deactivating one combination fails to mitigate the failure, we roll back the ineffective link deactivations and try deactivating the next combination.

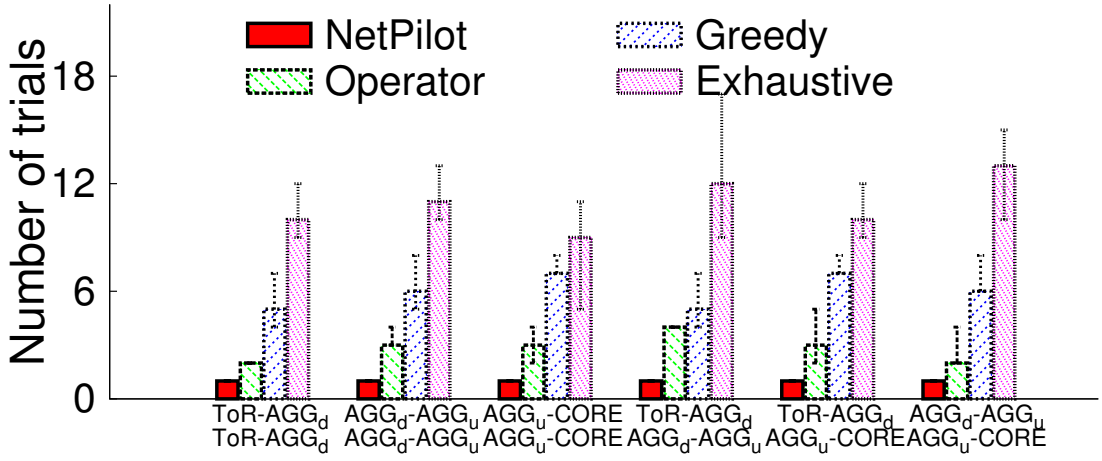


FIGURE 4.13: This figure compares the number of trials needed by different approaches to localize two simultaneously corrupted links in our testbed.

We design the first set of testbed experiments to show the number of attempts needed by each of the four algorithms to mitigate FCS errors. In each of the six experiments,

we pick two links in the testbed and use a commercial FCS error injector to corrupt 1% of all packets that traverse the two links. Two simultaneous faulty links in a large-scale DCN is common, as there are tens or hundreds of thousands of links. We repeat each experiment ten times using different traffic matrices. The histogram in Figure 4.13 shows the median number of trials for each experiment, while the error bars mark the maximum and minimum number of trials.

In almost all cases, NetPilot can accurately locate the two corrupted links in one trial. In two out of the sixty experiments, NetPilot identifies three links, the two malfunctioning links and one false positive link. This is far more effective than the operator's approach that will cumulatively deactivate two to five links before disabling the corrupted links. On the other hand, because the exhaustive search will roll back the ineffective link deactivations, it will eventually mitigate the FCS errors without having any healthy link deactivated. However, the disruption caused to the network by the tens of trials is significant.

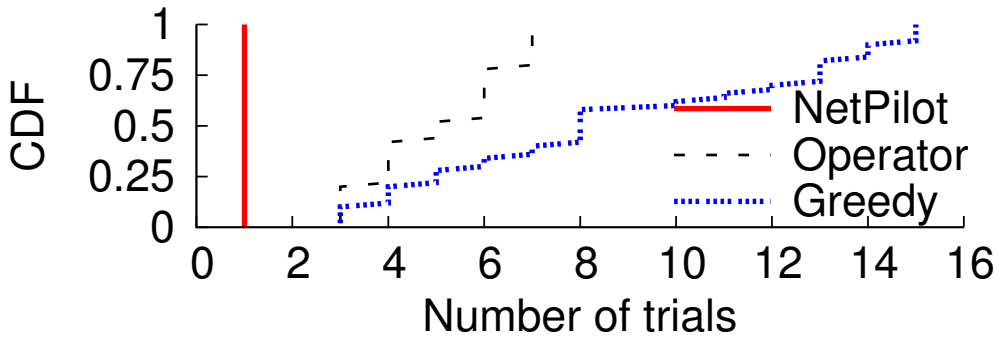


FIGURE 4.14: This figure shows the CDFs of the number of trials needed by different approaches to localize three simultaneously corrupted links. NetPilot significantly outperforms the others.

In the testbed experiments, we can only corrupt two links simultaneously due to the limitations of the FCS error injector. We use simulation to simulate more than two corrupted links in a much larger network, *i.e.*, DCN_p. We perform fifty simulations. In each simulation we randomly pick three links and set their corruption rates to be uniformly

distributed between 1% and 5% (typically observed in DCN$_p$). NetPilot's FCS model can uniquely localize the three corrupted links in 96% of the cases. For the remaining 4%, NetPilot localizes the three corrupted links plus one false positive link. Figure 4.14 shows that compared to NetPilot, other approaches require far more trials to mitigate just three simultaneously corrupted links. We omit the results from exhaustive search because its search space is too large and the simulations cannot finish in a reasonable amount of time.

To study the localization accuracy of NetPilot's FCS model in the real world, we replay the traffic matrices and switch counters from 78 DCNs$_p$'s FCS error instances on NetPilot. NetPilot can generate unique solutions and accurately localize the corrupted links in over 90% of the instances. In the remaining 10% of the instances, the solutions are not unique because the failure-time traffic matrices do not provide sufficient constraints for the linear equations, *e.g.*, we cannot tell if a link is corrupted when there is no traffic on it. For these cases, we pick the solution with the smallest number of non-zero corruption rates, as discussed in Chapter 4.4.3. We find this approach works well with the maximum one-link false positives.

4.6.4 Accurate Impact Estimation

In §4.4.2, we showed that NetPilot can accurately estimate link utilizations when no mitigation action is taken. We now conduct testbed experiments to show that NetPilot can accurately predict link utilizations as well as packet losses after device deactivations. We compare NetPilot with the operator's approach under five types of component deactivation: a randomly selected physical link, a LAG between an *AGG*$_d$ and an *AGG*$_u$, a LAG between an *AGG*$_u$ and a *CORE* switch, an *AGG*$_d$ switch, and an *AGG*$_u$ switch. For each deactivation type, we repeat the experiments under 144 different traffic matrices generated as follows. First, we collect the ToR-to-ToR traffic matrices from DCN$_p$ at a 10-minute granularity for one day. Then we use a modulo-16 hash function (16 is the number of *ToR*s in the testbed) to map the *ToR*s in DCN$_p$ to the *ToR*s in the testbed. Finally, we map the

ToR-to-ToR traffic matrices from DCN_p to the testbed and scale down the traffic volume by a scaling factor. We use a scaling factor that leads to no packet loss in the testbed to study how well NetPilot estimates link utilizations. We use a different scaling factor that leads to packet losses to study how well NetPilot estimates packet losses.

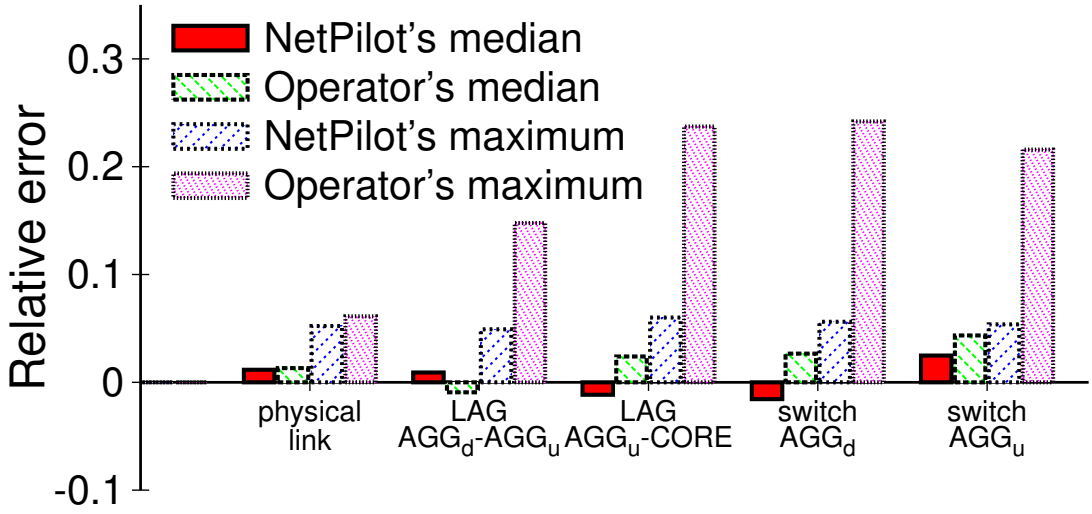


FIGURE 4.15: This figure compares the median and maximum relative errors of NetPilot's estimations of the maximum link utilization after deactivating various network components with those of the operator's manual approach. NetPilot has low estimation errors.

Figure 4.15 compares the relative errors of NetPilot's link utilization prediction with that of the operator's approach when there is no loss in the testbed. The relative error is defined as the difference between a predicted value and the value read from a switch counter normalized by a link's capacity: $\frac{prediction - switch\_counter}{link\_capacity}$.

NetPilot' median and maximum relative errors are less than 2.5% and 5% respectively. Although the median relative errors of the operator's approach are only slightly higher, its maximum relative errors can often exceed 20%. The main reason is that the operator's approach cannot predict the significant traffic load increase on links that are multiple hops away from a deactivated component.

Figure 4.16 is similar to Figure 4.15 except that we scale the traffic matrices to lead

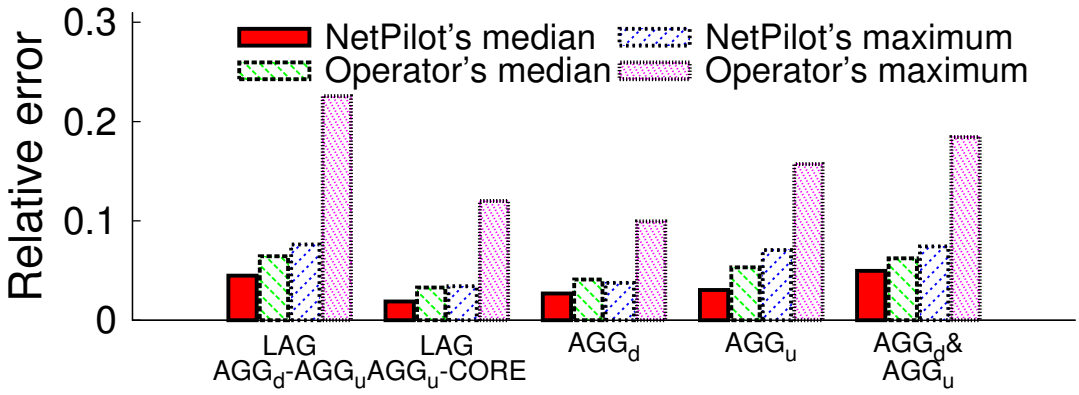


FIGURE 4.16: This figure compares the median and maximum relative errors of NetPilot's packet loss estimations with those of the operator's approach when deactivating various network components. Again NetPilot achieves low estimation errors.

to packet losses in the network. We also replace the case of deactivating a single physical link with the case of deactivating both an AGG_d and AGG_u, since the former rarely leads to packet losses. Again NetPilot's loss prediction has median and maximum relative errors below 5% and 8% respectively. Yet, the maximum relative error of the operator's approach is always more than double NetPilot's.

4.6.5 Effective Action Planning

When mitigating a failure, NetPilot must carefully choose the order of actions to minimize network disruption. In this section, we show that correctly ordering mitigation actions is challenging and often contradicts the operator's intuition.

We first compare NetPilot with the operator's approach when mitigating *uneven-split* failures. We collect all the link utilizations at a 10-minute granularity from DCN_p for one year and identify 151 *uneven-split* incidents by applying two criteria: 1) the difference between the maximum and mean link utilizations in the same LAG exceeds 5% (the same threshold used in §4.4.3); 2) the difference above lasts at least thirty minutes. Because some of these incidents did not cause link overload and thus were not investigated by oper-

ators, we do not know which component was responsible. Therefore, we randomly assign one "responsible" component to each incident and conduct heuristic-replay experiments.

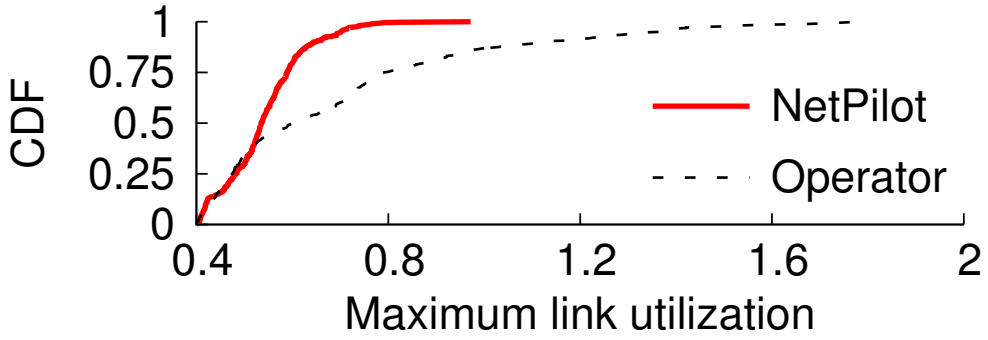


FIGURE 4.17: This figure compares the CDFs of the maximum link utilizations when NetPilot and operators restart components to mitigate the *uneven-split* failures. NetPilot causes lower maximum link utilizations during the failure mitigation period.

In these experiments, we emulate the operator's approach by first restarting LAGs and then switches (§4.6.1), whereas NetPilot exhaustively compares all possible actions under the latest TM and picks the one with the least impact (§4.4.3).

We use the CDFs of the maximum link utilizations under both approaches during the failure mitigation periods to compare NetPilot with the operator's approach, as shown in Figure 4.17. Because we randomly assign faulty components, it takes NetPilot and the operator's approach on average the same number of trials to successfully mitigate an *uneven-split* failure. Therefore, the failure mitigation periods under both approaches are roughly the same. The approach with lower maximum link utilizations is a better approach. For ease of presentation, we represent packet losses as link utilizations greater than 100%. As can be seen, for 60% of the incidents, NetPilot is far less disruptive than the operator's approach. NetPilot never overloads links while the operator's approach would lead to traffic losses in 20% of the incidents.

Unlike *uneven-split* failures, *link-down* failures are mitigated by deactivations in which the deactivated components may remain down for days. The operator's approach uses the

latest TM right before the actions to estimate the impact of deactivations, and to choose the actions with the least negative impact. NetPilot has two main advantages over this approach: 1) it can use multiple TMs that better approximate future TMs in the deactivation periods to estimate impact; and 2) its impact estimation is more accurate (§ 4.6.4).

Although we have shown NetPilot's impact estimation is more accurate in the previous subsection, we further use a failure-replay experiment to show that more accurate impact estimation can lead to less disruptive mitigation actions. We replay 97 *link-down* incidents observed in DCNs$_p$ during one year. For each incident, NetPilot uses the same TM as the operator does to estimate the impact of a deactivation action. We then compare the resulting maximum link utilizations right after NetPilot's deactivation actions with those from the real failure traces.

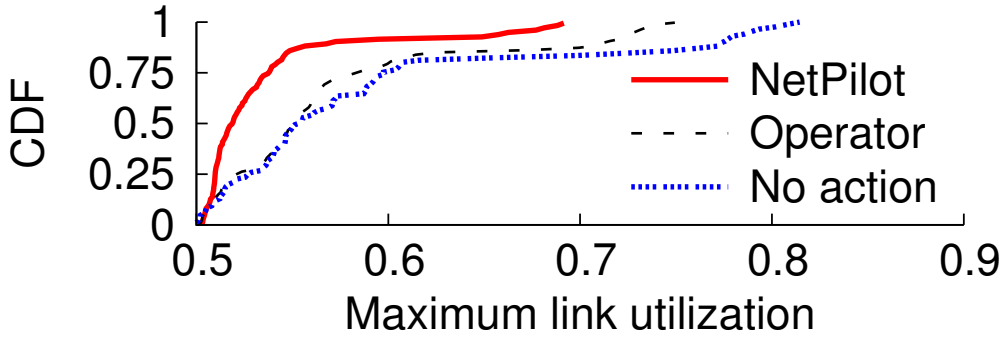


FIGURE 4.18: This figure shows the CDFs of the maximum link utilizations right after each method takes its deactivation action to mitigate *link-down* failures in DCN$_p$. NetPilot outperforms other methods because its impact estimation is more accurate.

Figure 4.18 shows the comparison results. As can be seen, NetPilot's actions are noticeably better than the actual actions taken by the operators or taking no action at all. These results suggest that NetPilot's higher estimation accuracy can lead to less disruptive mitigation actions.

Next, we run testbed experiments to show that using multiple TMs to approximate future TMs during a device's deactivation period can also lead to less disruptive mitigation

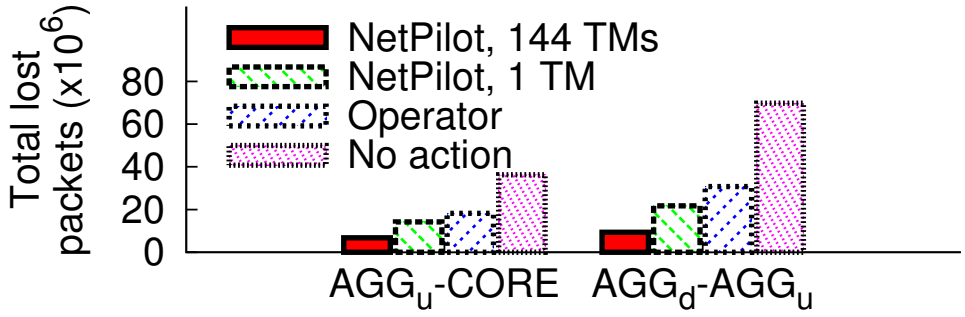


FIGURE 4.19: This figure compares the accumulated packet losses in a 24-hour period right after a deactivation action is taken in our testbed. When using multiple historical TMs to approximate future TMs, NetPilot leads to the fewest packet losses.

actions. In these experiments, we inject *link-down* failures in the testbed by shutting down half of the physical links in a LAG either between an AGG_d and an AGG_u or between an AGG_u and a *CORE*. We then use the 144 TMs from the preceding 24 hours, rather than just the TM right before a deactivation action, to plan mitigation actions. We then measure the packet losses during the following 24 hours using the TMs in those hours.

Figure 4.19 presents the packet losses under each method. As can be seen, NetPilot has the fewest packet losses when using multiple historical TMs to approximate future TMs.

4.7 Conclusion

NetPilot is a system that automatically mitigates DCN failures. It is a departure from the status quo that relies heavily on human intervention. We believe that our work is critical to managing modern DCNs given the ballooning number of devices in these DCNs and the trend towards commodity hardware. NetPilot works by identifying a candidate set of afflicted components that are likely to cause a problem and iteratively taking mitigation actions targeting each one until the problem is alleviated. A key insight that makes this approach viable is the redundancy presented in modern DCN topologies. This redundancy reduces the potential for any single deactivated or rebooted component to disrupt a net-

work. Our experiments show that NetPilot can successfully detect and mitigate several common types of failures both in a testbed and in a real production DCN.

5

Conclusion

In conclusion, we target two primary challenges in managing modern datacenters: how to design a practical datacenter network architecture and how to achieve fast network failure recovery. We propose DARD and MPXCP to solve the first challenge and NetPilot to solve the second challenge. In this chapter, we summarize the key contributions of the thesis.

To the best of our knowledge, DARD is the first distributed adaptive routing architecture for datacenters. Our contributions are: 1) We justify four design guidelines for practical datacenter architectures and explain the pros and cons of existing solutions. 2) We design DARD, which is proven to converge to a Nash equilibrium in finite steps and with bounded gap to the optimal flow allocation. 3) We conduct intensive evaluation in both testbed and simulation to show DARD's quick convergence, high bisection bandwidth and small control overhead.

To the best of our knowledge, MPXCP is the first explicit multipath congestion control protocol. Our contribution lies in three aspects: 1) We discover MPTCP's performance degradation in four aspects: slow convergence, inefficiency, being biased against long-RTT flows and large queue size. 2) We explore the design space to address MPTCP's four performance degradations. We further design MPXCP to efficiently and fairly utilize the

capacity despite different delay and bandwidth products. 3) We implement MPXCP in *ns-2*. Through intensive evaluation, we confirm that MPXCP outperforms MPTCP in the face of large delay-bandwidth product.

To the best of our knowledge, NetPilot is the first automated failure mitigation system for datacenter networks. Our contributions are: 1) We study and classify the high-impact failures in production datacenter networks over a six-month period, and find that we can mitigate most of those failures by simple actions such as restart or deactivation. We also find that there is sufficient redundancy in a DCN to accommodate the impact of mitigation actions. 2) We design and implement NetPilot and deploy it in a testbed that resembles a real datacenter network topology. We also conduct simulations using data from production to evaluate NetPilot at a large scale. We experimentally validate the accuracy of the Impact Estimator, and find that it offers an error rate of less than 8%. 3) We use NetPilot to automatically mitigate three types of high-impact failures that operators often encounter in production DCNs. Besides reducing operational overhead, NetPilot decreases the median mitigation time from 2 hours to 20 minutes compared to current operational practice, significantly shortening a failure's impact on online services. 4) We justify the design choices made in NetPilot. Compared to simple heuristics, NetPilot can succeed with fewer trials while maintaining safe operating conditions in the network.

Bibliography

(2002), "Management Information Base (MIB) for the Simple Network Management Protocol (SNMP)," United States, RFC Editor.

(2008), "Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments," .

(2012), "Virtual PortChannels: Building Networks without Spanning Tree Protocol," .

Al-Fares, M., Loukissas, A., and Vahdat, A. (2008), "A scalable, commodity data center network architecture," in *SIGCOMM '08*.

Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. (2010), "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. (2010), "Data Center TCP DCTCP," in *SIGCOMM'10*.

Amazon (2011), "Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region," .

Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. (2007), "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *SIGCOMM '07*.

Bennett, J. C. R. and Zhang, H. (1996), "Hierarchical packet fair queueing algorithms," in *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '96, pp. 143–156, New York, NY, USA, ACM.

Benson, T., Akella, A., and Maltz, D. A. (2010), "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th annual conference on Internet measurement*, pp. 267–280.

Boudec, J.-Y. (2000), "Rate adaptation, Congestion Control and Fairness: A Tutorial," .

Busch, C. and Magdon-Ismail, M. (2009), "Atomic routing games on maximum congestion," vol. 410, pp. 3337–3347, Essex, UK, Elsevier Science Publishers Ltd.

Chen, Y., Griffith, R., Liu, J., Katz, R. H., and Joseph, A. D. (2009), "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, WREN '09, pp. 73–82, New York, NY, USA, ACM.

Chiu, D.-M. and Jain, R. (1989), "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," vol. 17, pp. 1–14, Amsterdam, The Netherlands, The Netherlands, Elsevier Science Publishers B. V.

Cisco Systems, Inc. (2011), "Cisco Data Center Infrastructure 2.5 Design Guide," .

Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalag, P., Sharma, P., and Banerjee, S. (2011), "Devoflow: Scaling Flow Management for High-Performance Networks," in *SIGCOMM'11*.

Dally, W. and Towles, B. (2003), *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Floyd, S. and Jacobson, V. (1993), "Random early detection gateways for congestion avoidance," vol. 1, pp. 397–413, Piscataway, NJ, USA, IEEE Press.

Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. (2009), "VL2: a scalable and flexible data center network," in *SIGCOMM '09*.

Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. (2008), "NOX: towards an operating system for networks," vol. 38, pp. 105–110, New York, NY, USA, ACM.

Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., and McKeown, N. (2012), "Where is the debugger for my software-defined network?" in *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pp. 55–60, New York, NY, USA, ACM.

Hoelzle, U. and Barroso, L. A. (2009), *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan and Claypool Publishers.

Hong, C.-Y., Caesar, M., and Godfrey, P. B. (2012), "Finishing flows quickly with preemptive scheduling," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pp. 127–138, New York, NY, USA, ACM.

Hopps, C. (2000), "Analysis of an Equal-Cost Multi-Path Algorithm," No. 2992 in Request for Comments, IETF.

IEEE (2000), "802.3ad Link Aggregation Standard," IEEE standard.

Kandula, S., Katabi, D., and Vasseur, J.-P. (2005a), "Shrink: a tool for failure diagnosis in IP networks," in *MineNet '05*.

Kandula, S., Katabi, D., Davie, B., and Charny, A. (2005b), "Walking the tightrope: Responsive yet stable traffic engineering," in *In Proc. ACM SIGCOMM*.

Kandula, S., Mahajan, R., Verkaik, P., Agarwal, S., Padhye, J., and Bahl, P. (2009a), "Detailed diagnosis in enterprise networks," in *SIGCOMM '09*.

Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R. (2009b), "The nature of data center traffic: measurements & analysis," in *IMC '09*.

Katabi, D., Handley, M., and Rohrs, C. (2002), "Congestion control for high bandwidth-delay product networks," vol. 32, pp. 89–102, New York, NY, USA, ACM.

Khanna, A. and Zinky, J. (1989), "The revised ARPANET routing metric," in *Symposium proceedings on Communications architectures & protocols*, pp. 45–56, ACM.

Kompella, R.R and Yates, Jennifer, and Greenberg, Albert and Snoeren, Alex (2005), "IP Fault Localization Via Risk Modeling," in *NSDI '05*.

Lab, D. (2003), "Deter Lab," .

Lakshman, M. K., Lakshman, T. V., and Sengupta, S. (2004), "Efficient and Robust Routing of Highly Variable Traffic," in *In Proceedings of Third Workshop on Hot Topics in Networks (HotNets-III)*.

Lonvick, C. (2001), "The BSD Syslog Protocol," United States, RFC Editor.

Meeker, M. and Wu, L. (2013), "Internet Trends," .

Niranjan Mysore, R., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., and Vahdat, A. (2009), "PortLand: a scalable fault-tolerant layer 2 data center network fabric," in *SIGCOMM '09*.

Nishida, Y. (2010), "ns-2 implementation of MPTCP," .

Openflow (2008), "OpenFlow 1.0," .

Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. (2012), "FairCloud: sharing the network in cloud computing." in *SIGCOMM*, pp. 187–198, ACM.

Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. (2011), "Improving datacenter performance and robustness with multipath TCP," vol. 41, pp. 266–277, New York, NY, USA, ACM.

Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. (2012), "How hard can it be? designing and implementing a deployable multipath TCP," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pp. 29–29, Berkeley, CA, USA, USENIX Association.

S. Nadas, Ericsson (2010), "Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6," .

Shaikh, A., Isett, C., Greenberg, A., Roughan, M., and Gottlieb, J. (2002), "A case study of OSPF behavior in a large enterprise network," in *IMW '02*.

Sironi, G. (2001), "IP Aliasing HowTo," .

Steve (2005), "TCPTrack," .

T. Li, B. Cole, P. Morton, D. Li (1998), "Cisco Hot Standby Router Protocol (HSRP)," No. 2281 in Request for Comments, IETF.

Wischik, D., Raiciu, C., Greenhalgh, A., and Handley, M. (2011), "Design, implementation and evaluation of congestion control for multipath TCP," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*.

Wu, X. and Yang, X. (2012), "DARD: Distributed Adaptive Routing for Datacenter Networks," in *Proceedings of The 32nd International Conference on Distributed Computing Systems (ICDCS)*, Macau, China.

Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., and Zhang, M. (2012), "NetPilot: Automating Datacenter Network Failure Mitigation," in *SIGCOMM'12*.

Yang, X., Clark, D., and Berger, A. W. (2007), "NIRA: A New Inter-Domain Routing Architecture," vol. 15.

Biography

My name is Xin Wu, born on 06/23/1985 at Yuci Shanxi, China. I am a PhD student at Duke University Computer Science Department from 08/2009 to 12/2013, working with Professor Xiaowei Yang. I got my Master of Computer Science from Duke University in 2013. Before my study at Duke, I received Master and Bachelor from Tsinghua University, Beijing China in 2009 and 2007 respectively, majored in Computer Science. During my PhD study, I received NSDI travel grant and Outstanding PhD Initiation Project Award in 2011. I will join Big Switch Networks (www.bigswitch.com) as a technical stuff after my graduation.

Following are my publications by 08/2013.

Hongqiang Harry Liu, **Xin Wu**, Ming Zhang, Lihua Yuan, Roger Wattenhofer, Dave A. Maltz, zUpdate: Updating Data Center Networks with Zero Loss, SIGCOMM 2013

Yu Chen, **Xin Wu**, Qiang Cao, Xiaowei Yang, BigPi: Sharing Link Pools in Cloud Networks, Duke University technical report, 2013

Xin Wu, Daniel Turner, George Chen, Dave Maltz, Xiaowei Yang, Lihua Yuan, Ming Zhang, NetPilot: Automating Datacenter Network Failure Mitigation, SIGCOMM 2012

Xin Wu and Xiaowei Yang, DARD: Distributed Adaptive Routing for Datacenter Networks, ICDCS 2012

Xin Wu and Xiaowei Yang, DARD: Distributed Adaptive Routing for Datacenter Networks, Post session, NSDI 2011

Yu Chen **Xin Wu** Xiaowei Yang, MAPS: Adaptive Path Selection for Multipath Trans-

port Protocols in the Internet, Duke University technical report, 2011

Xia Yin, **Xin Wu**, Kilnam Chon, Zhiliang Wang. ISPSG: Internet Service Provider-Separated Geographic-Based Addressing and Routing. 2nd International Workshop on the Network of the Future (FutureNet 2009), Honolulu, Hawaii. December 2009

Zhiliang Wang, Xia Yin, Yang Xiang, Ruiping Zhu, Shirui Gao, **Xin Wu**, Shijian Liu, Song Gao, Li Zhou, Peng Li. TTCN-3 Based Conformance Testing of Mobile Broadcast Business Management System in 3G networks. 21st IFIP Int. Conference on Testing of Communicating Systems and the 9th Int. Workshop on Formal Approaches to Testing of Software (Testcom/FATES 2009). Eindhoven, the Netherlands. November, 2009

Xia Yin, **Xin Wu**, Zhiliang Wang. Measuring IP Address Fragmentation from BGP Routing Dynamics. The 4th International ICST Conference on Scalable Information Systems (INFOSCALE 2009). Hong Kong, China. June, 2009

Xin Wu, Xia Yin, Zhiliang Wang, Min Tang. A three-step dynamic threshold method to cluster BGP updates into routing events. The 9th International Symposium on Autonomous Decentralized Systems (ISADS 2009). Athens, Greece. March, 2009